InterConnect
2017

March 19–23
MGM Grand &
Mandalay Bay
Las Vegas, NV

IBM

Hands-on Lab

Session 2166

IBM Integration Bus and
IBM Integration Bus on Cloud

RESTAsyncRequest node
KafkaProducer node
LoopBackRequest node
Callable Flows
Docker
IIB on Cloud

Provided by IBM BetaWorks

Table of Contents

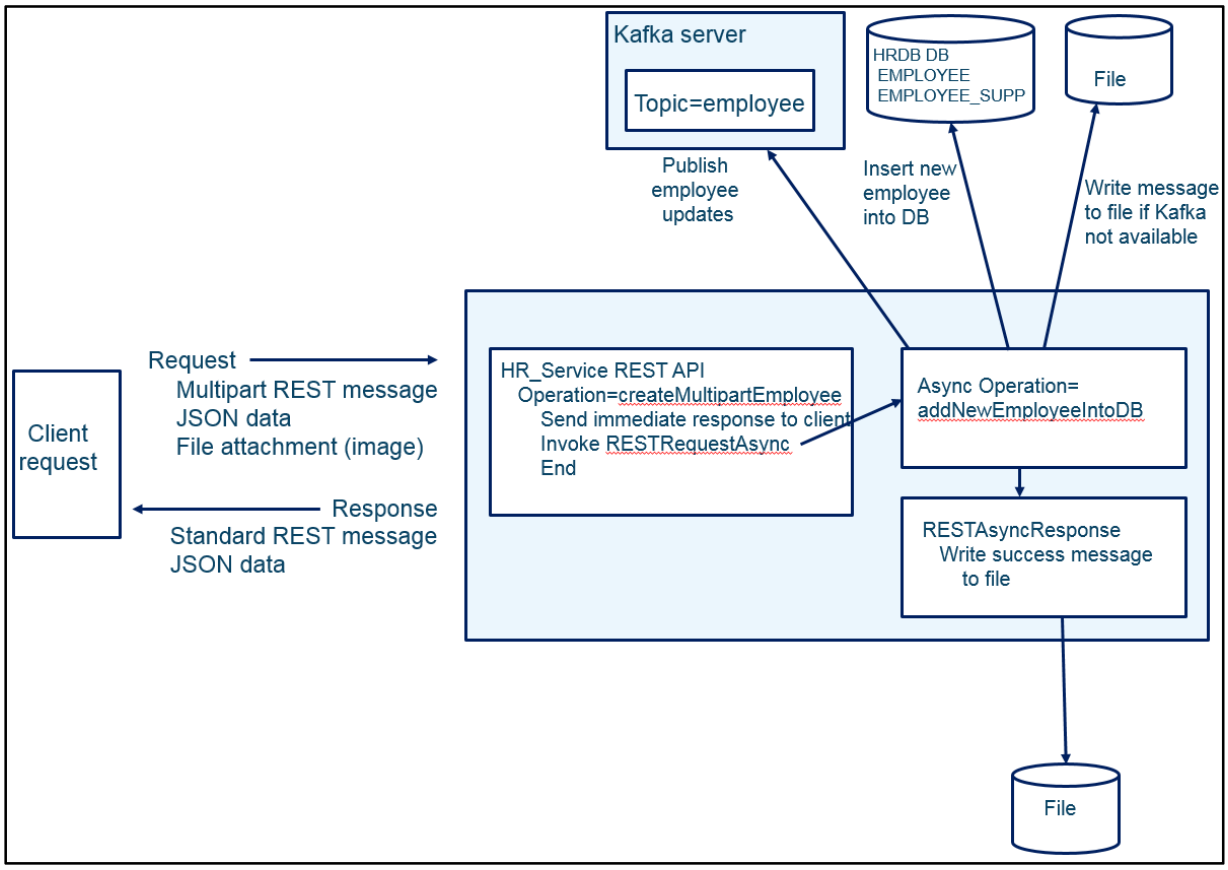
1. Part 1 – Asynchronous REST Request node

1.1 Outline of integration

The first part of this lab session provides a technique for an IIB REST API to receive a REST request that contains a multipart message payload in MIME format. The scenario was constructed to address the requirement to process a REST message that contained a binary image, as well as the standard JSON payload.

In this case, the REST message contains a standard JSON-formatted message, but also includes an attachment. The REST message contains “new employee” data, and the IIB REST API will take this information and add a new row to the EMPLOYEE and EMPLOYEE_SUPPLEMENTARY tables. The attached file (an image of the new employee) is included in the data written to the database.

Any updates made to the HRDB database are then sent to the local Kafka server, using the KafkaProducer node in the IIB operation.



This integration scenario shows you how to use the REST Request node to invoke a REST API operation. The REST Request node can be used synchronously or asynchronously. This scenario will demonstrate the use of the **asynchronous** flavour of this node.

The scenario implements a REST API, HR_Service, that processes a REST request that has an attachment. HR_Service performs some initial processing, and then sends an immediate response back to the client. HR_Service then uses the RESTAsyncRequest node to invoke a further REST operation, where the main processing logic is performed.

The requesting client sends a REST request that contains two parts:

- A standard REST message in JSON format
- A binary part, in the form of a jpg image, attached to the REST request as a file.

In detail, the request message contains:

Part 1 - employeeData

```
{
"EMPLOYEE": {
        "EMPNO": "000003",
        "FIRSTNME": "Albert",
        "MIDINIT": "J",
        "LASTNAME": "Einstein",
        "WORKDEPT": "A00",
        "PHONENO": "0101",
        "HIREDATE": "1912-07-27",
        "JOB": "Manager",
        "EDLEVEL": 9,
        "SEX": "M",
        "BIRTHDATE": "1879-03-14",
        "SALARY": 9990,
        "BONUS": 4440,
        "COMM":6660 }
“EMPLOYEE_SUPPLEMENTARY”: {
        “EMAIL”: userid@domain.com,
        “MOBILEPHONE”: “447878123456”,
        “TWITTERID”: “@davidh”,
        “BOXID”: username@domain.com,
        “IMAGE”: “Used only for the embedded image version of this app”}
}
```

Part 2 - employeeImage

Image of employee in binary format – used with the File Attachment version of this app

1.1.1 Tasks

In this part of the lab, you will perform the following tasks:

- Import and explore a partially completed REST API and IIB Application
- Complete the REST API and Application by adding the RESTAsyncRequest and Response nodes, and the KafkaProducer node.
- Deploy the completed applications, and test with the Chrome Postman tool
- (Optionally) – rerun the tests using the debug feature of IIB.

1.1.2 Preparation

This scenario is based on the solution of the REST API HR_Service. The scenario uses an IIB node called TESTNODE_iibuser. This has already been created and configured to support this scenario.

This scenario does not ask you to build the solution from scratch. A complete solution is provided, and you will investigate various aspects of the solution, and perform a test of the provided solution.

1.1.3 The Chrome Postman plugin

Because this scenario needs to send a REST request with a mixed format message payload, you will need to use a test tool that is capable of generating such a request. In the development of this scenario, we have used Chrome Postman for this. This is already installed on your system.

Note that when Postman executes, this app does not actually run under Chrome; it executes as a stand-alone application.

1.1.4 Model Definitions

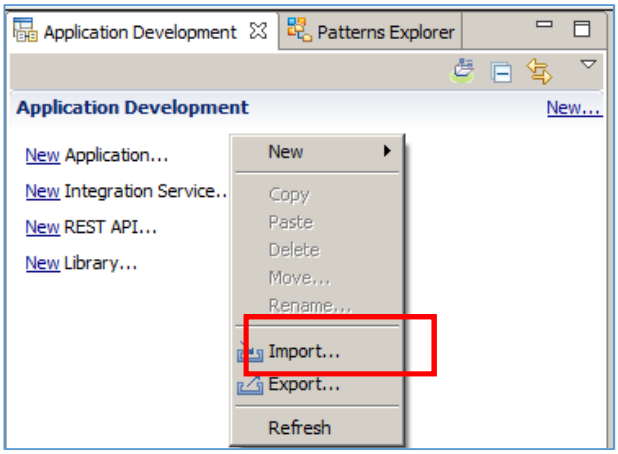
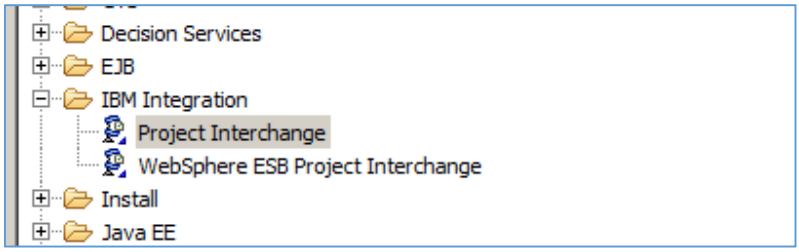
The following JSON message models are used by this version of the HR_Service REST API.

- DBRESP – contains database response information
- EMPLOYEE – defines columns in the EMPLOYEE table
- EMPLOYEE_SUPPLEMENTARY – defines columns in the EMPLOYEE_SUPPLEMENTARY table
- DEPARTMENT – defines columns in the DEPARTMENT table
- EmployeeResponse
  - DBResp (type = DBRESP)
  - Employee (Array, type = EMPLOYEE)
- DepartmentResponse
  - DBResp (type = DBRESP)
  - Department (Array, type = DEPARTMENT)
- CompleteResponse
  - DBResp_employee (type = DBRESP)
  - Employee (type = EMPLOYEE, single object, not array)
  - DBResp_department (type = DBRESP)
  - Department (type = DEPARTMENT, single object, not array)
  - DBResp_employee_supplementary (type = DBRESP)
  - Employee_supplementary (type = EMPLOYEE_SUPPLEMENTARY)
- EmployeeSupplementaryResponse used when only accessing EMPLOYEE_SUPPLEMENTARY)
  - DBResp
  - Employee_supplementary
- EmployeeAddUpdateCompleteRequest (input message, used when adding a complete new employee)
  - Employee
  - EmployeeSupplementary

1.2 Explore the HR_Service REST API

In this section of the lab exercise, you will import and explore the partial solution provided for this scenario.

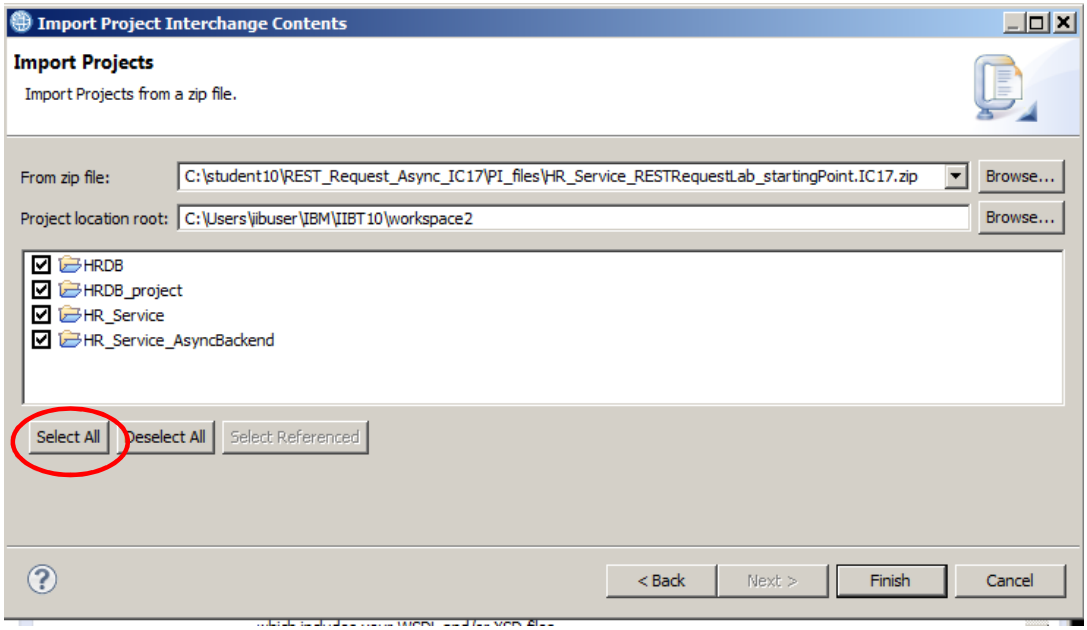
1.2.1 Import the IIB REST API

1.	Ensure you are logged in to Windows as the user "iibuser", password = "passw0rd". (You may already be logged in). If it's not started already, start the IIB Toolkit from the Start menu.
2.	To avoid naming clashes, this scenario will be developed using a new workspace. In the Integration Toolkit, click File, Switch Workspace. Give the new workspace the name "RESTRequest", or similar.
3.	Right-click in the Application Development pane and click 'Import':  In the IBM Integration folder, select Project Interchange, and click Next. 

4. Import the following Project Interchange (PI) zip file:

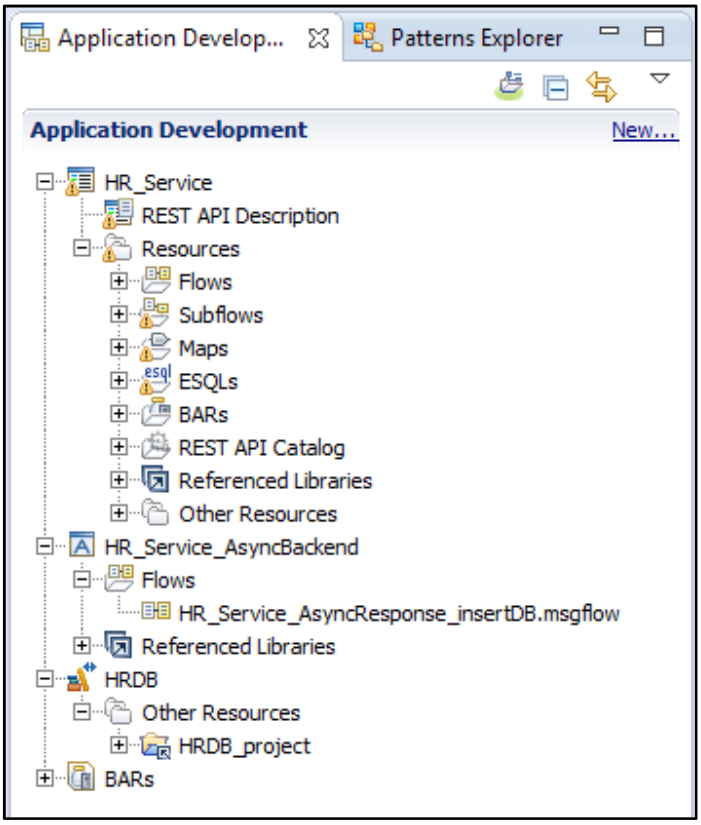
`C:\student10\REST_Request_Async_IC17\PI_files\`
`HR_Service_RESTRequestLab_startingPoint.IC17.zip`

Note: Make sure that all four projects in this PI file are selected for import. The PI includes the HRDB shared library and database project.



5. When imported, you should have in your workspace the **HR_Service** REST API and the **HRDB** shared library that is referenced by the REST API.

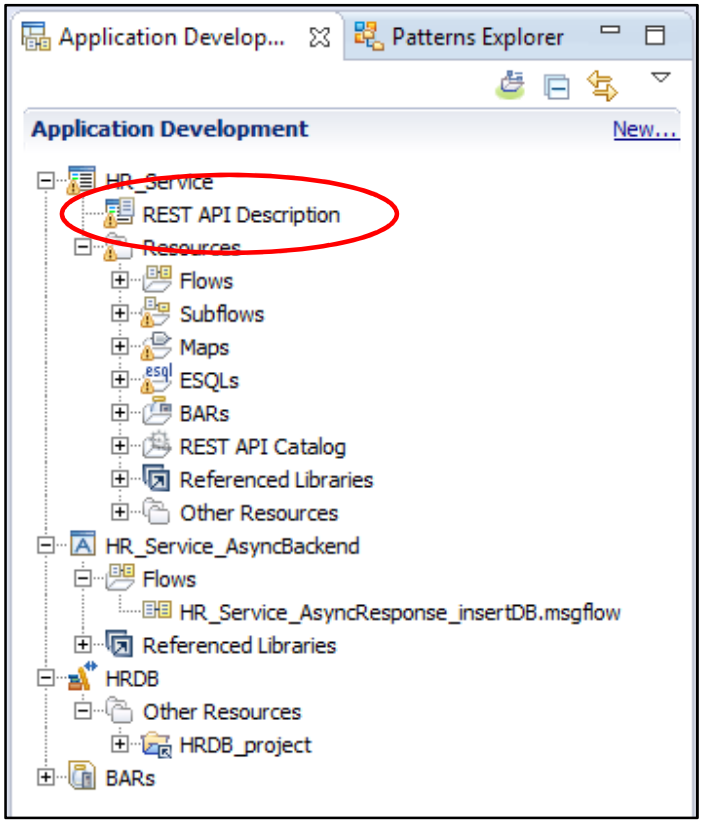
   The application **HR_Service_AsyncBackend** contains the message flow where you will add theRESTAsyncResponse node, that will be invoked by the HR_Service operation.



1.2.2 Explore HR_Service

In this section, you will explore the imported REST API.

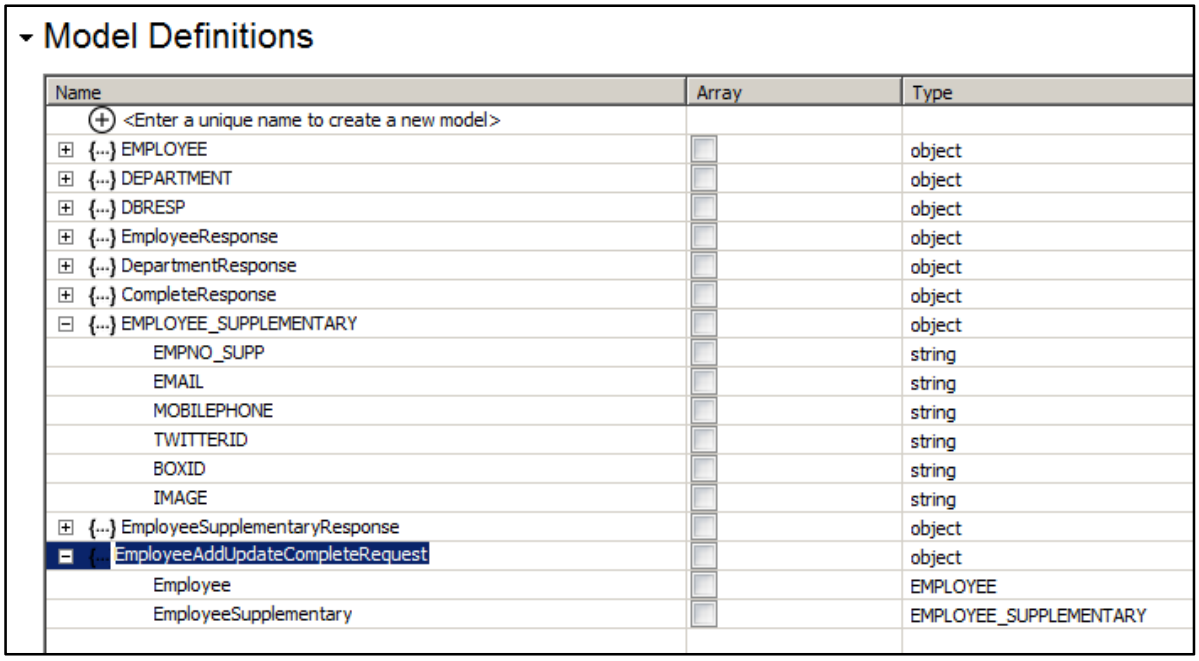
1. Expand the HR_Service REST API and double-click “REST API Description”.



2. In the main editor view, scroll down to the Model Definitions section. Expand Model Definitions, and expand some of the models.

   The EmployeeAddUpdateCompleteRequest model is used in this lab. It comprises two element types, EMPLOYEE and EMPLOYEE_SUPPLEMENTARY, each of which have their own model definitions, shown in the list.

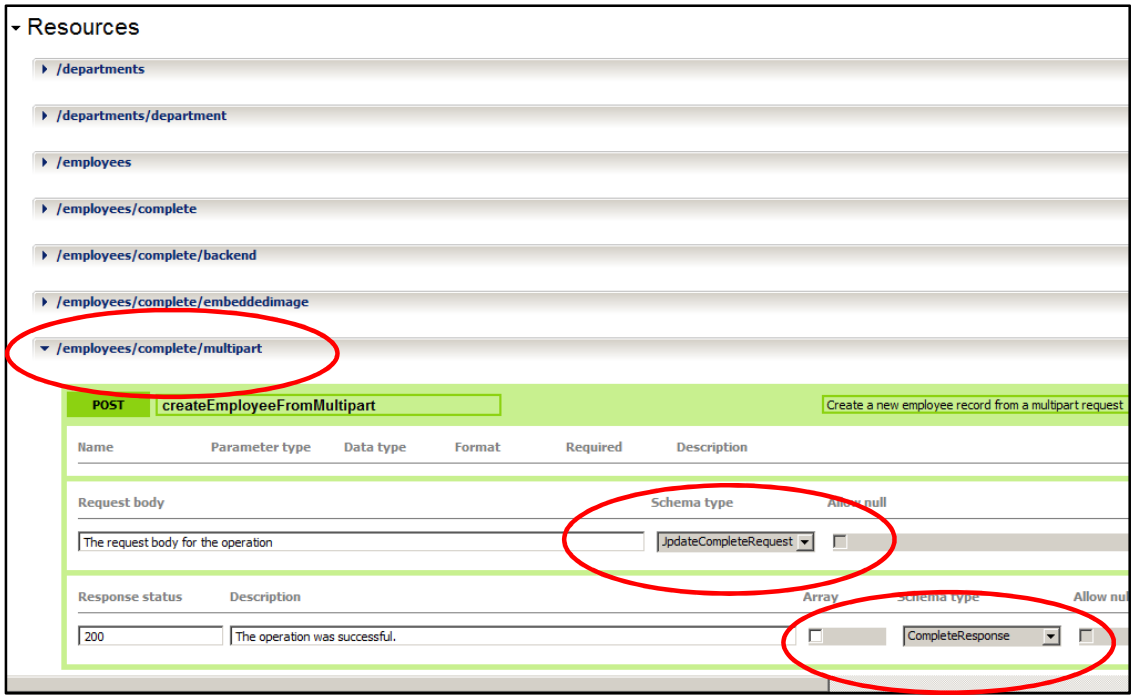
   When you have finished here, collapse the Model Definitions section.



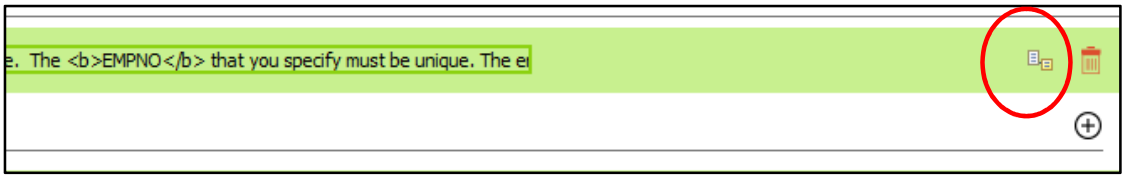
3. Move up to the Resources section, and expand the **employees/complete/multipart** resource.

   Look at the **createEmployeeFromMultipart** POST operation.

   Note that the Request body has a schema type of **EmployeeAddUpdateCompleteRequest**, and the Response body has a schema type of **CompleteResponse**.



4. Scroll to the right, and open the subflow implementation of this operation.



5. This opens the implementation subflow. Each node will be investigated in detail over the next few pages.

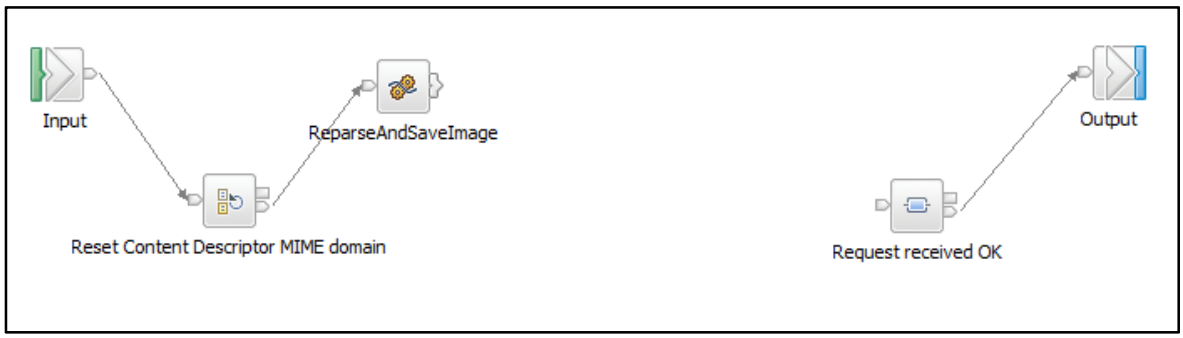
   At a high level, the subflow performs the following actions:

   a. Because the application is a REST API, the default parser has been set to JSON. Therefore, initially, the subflow will not be able to properly parse the incoming message (which is a MIME multipart message). To handle this, the first processing node in the subflow is a "Reset Content Descriptor" node, which will reparse the message using the MIME domain. When this happens, the multipart message is parsed into its two constituent parts, Part1 (the JSON data, but still held in binary format at this stage) and Part2 (the binary image).

   b. The ReparseAndSaveImage node is an ESQL Compute node which further processes these two parts.
      - Part 2 (the image) is stored in the IIB Environment tree, and then converted to Base64 encoded and stored back in the message tree.
      - Part 1 is reparsed using the JSON parser, and the resulting parsed data is stored in the message tree, now in JSON format.

   c. The "Request received OK" mapping node constructs a simple response message to acknowledge receipt of the message for the client.

   The empty space below is where you will later add a new RESTAsyncRequest node.



1.2.3 Explore the REST API in detail

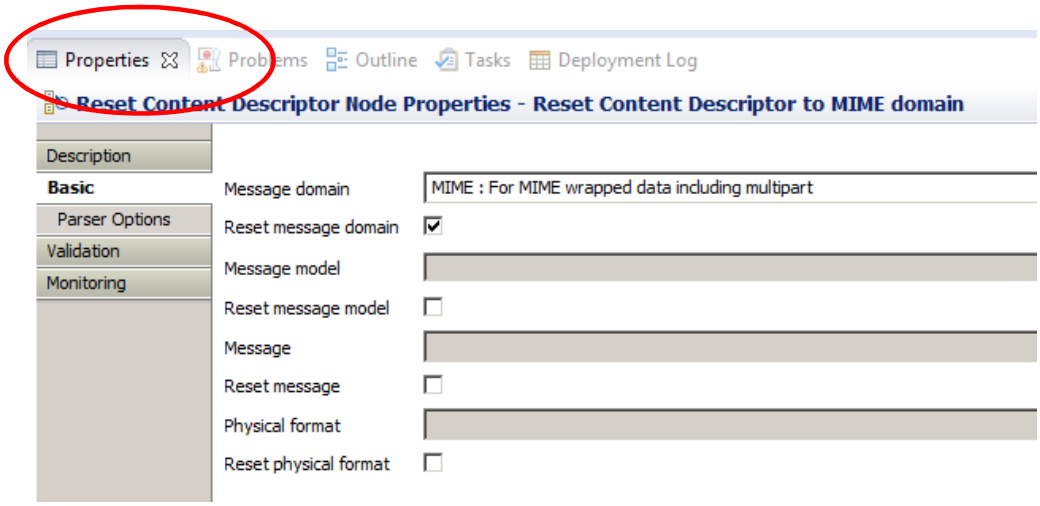
1. Select (click on) the Reset Content Descriptor node, and highlight the node Properties.

   On the Basic tab, the following properties have been set:

   - Message domain = MIME: for MIME wrapped data including multipart
   - Reset message domain = ticked

   This means that the incoming message, which is initially in MIME format, will be reparsed as a MIME multipart message (the default parser for a REST API is JSON). When this is done, the message tree will consist of several message parts, as many parts as exist in the incoming message. In this example, you will see two message parts (because the REST message is assumed to have just one attachment in this scenario).

   Note that subsequent nodes in the flow can address different parts of the message only by using the MIME parser, unless the message is again reparsed.



2. Open the ESQL Compute node **ReparseAndSaveImage.**

At this point in the flow, the message tree has been parsed by the MIME parser and split into MIME "Parts". Each part is referenced by the element name "Part[n]", so is referenced like this:

OutputRoot.MIME.Parts.Part[n]

The raw data (BLOB) is referenced by the value
OutputRoot.MIME.Data.BLOB.BLOB

So, the following statement adds a new element called "Data", under the "Part[1]" element, and populates the contents of "Data" by using the PARSE option with the input as shown here. Note that 546 represents the Encoding of the data and 1208 represents the CCSID.

```
CREATE LASTCHILD OF OutputRoot.MIME.Parts.Part[1].Data
     DOMAIN('JSON')
     PARSE(OutputRoot.MIME.Parts.Part[1].Data.BLOB.BLOB, 546,
            1208);
```

The next statement sets the "Data.BLOB" portion of the Part[1] output message to Null. This is required because we no longer need the BLOB form of the message.

```
SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;
```

This statement saves the entire contents of Part[2] (the binary image) to the Environment tree (in a folder called Variables).

```
set Environment.Variables.Image =
                    OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;
```

The next two statements manipulate the Employee and EmployeeSupplementary parts of the message, in JSON format. The OutputRoot.JSON.* parts of the message are constructed, by extracting them from the MIME part of the message.

```
-- Set the Message Tree - Employee element
set OutputRoot.JSON.Data.Employee =
       OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.Employee;

-- Set the Message Tree - EmployeeSupplementary element
set OutputRoot.JSON.Data.EmployeeSupplementary =
       OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.EmployeeSupplementary;
```

This statements converts the IMAGE (previously stored in the Environment.Variables tree) into a Base64 encoded version, and stores the converted element into the OutputRoot tree, now fully parsed and held in the JSON domain.

```
-- Make special arrangements for the binary image, which we want to
   store in the database as Base64 encoded.
Set OutputRoot.JSON.Data.EmployeeSupplementary.IMAGE =
                                    BASE64ENCODE(Environment.Variables.Image);
```

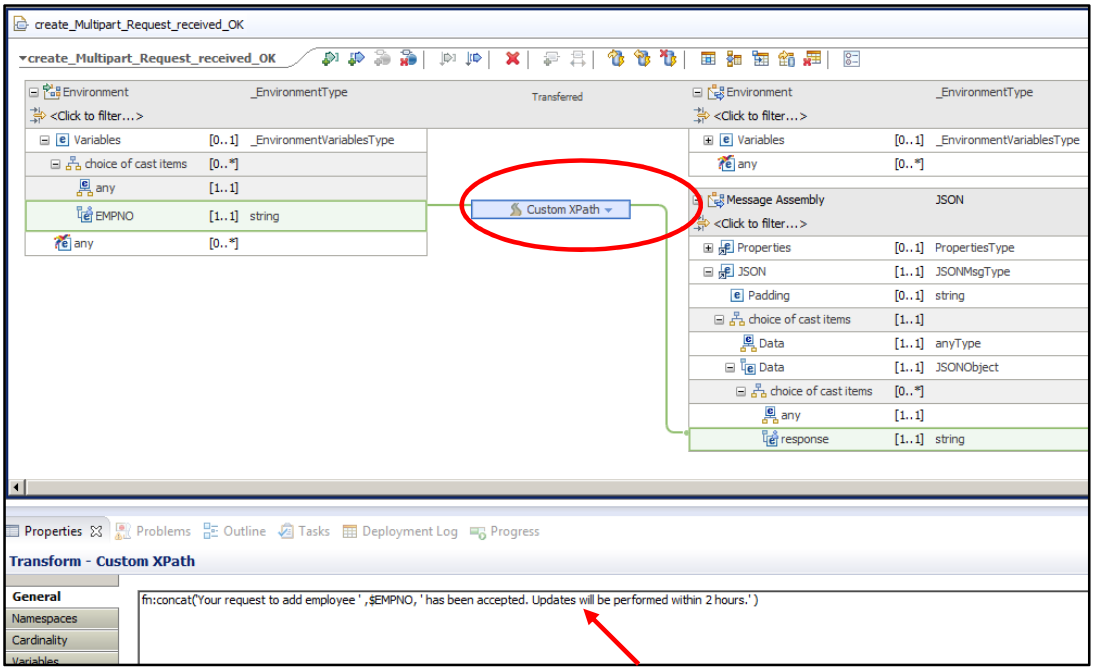
This statement saves the value of EMPNO in the Environment.Variables tree, so that an appropriate response can be returned to the client.

```
-- Save EMPNO in Env, so that the later map can send an appropriate
   message back to the client.
Set Environment.Variables.EMPNO = OutputRoot.JSON.Data.Employee.EMPNO;
```

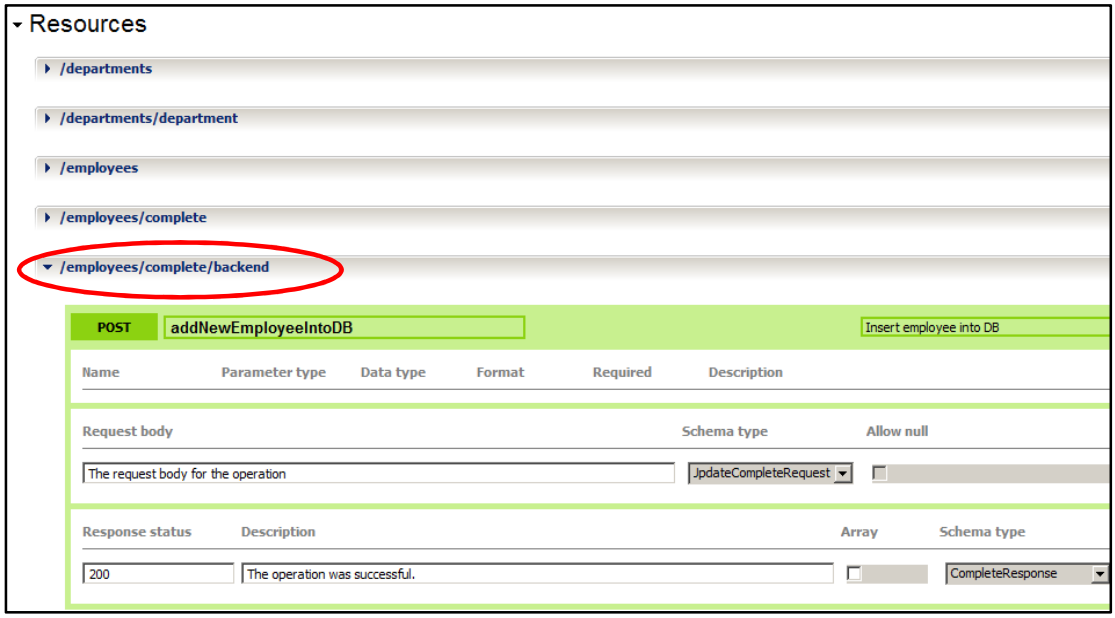
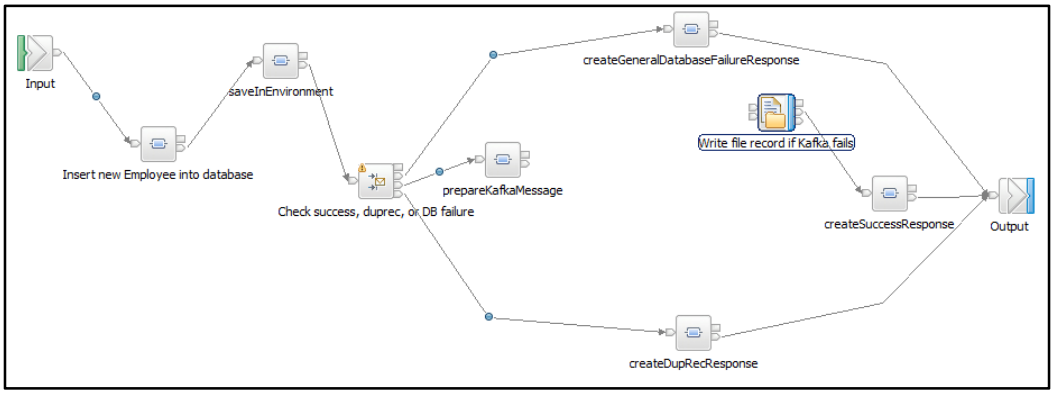
And finally this statement set the value of the MIME part of the message tree to null.

```
set OutputRoot.MIME = null
```

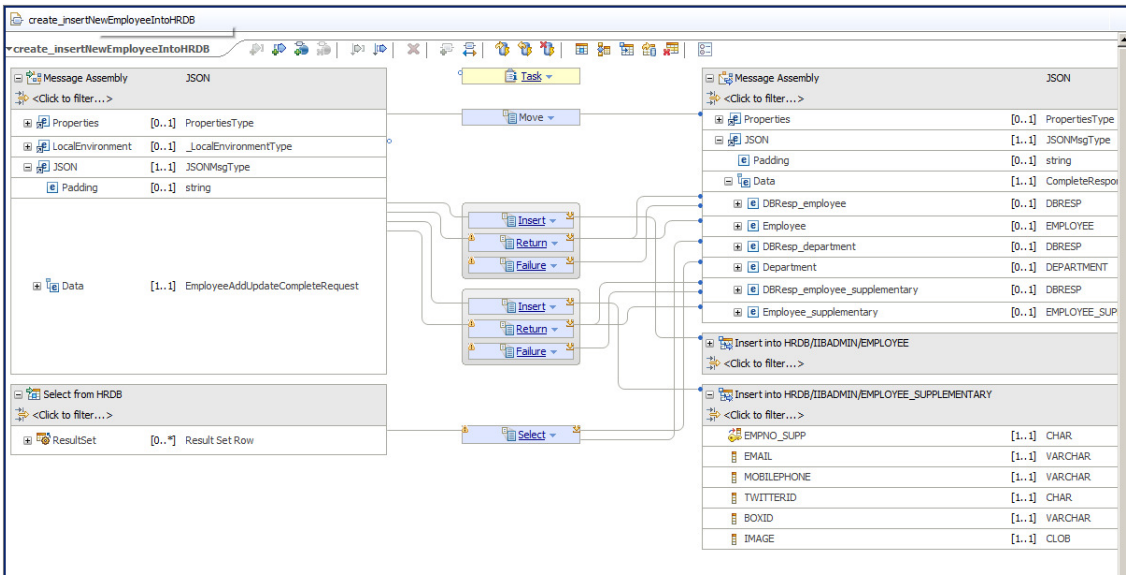
Close the Compute node.

3.	The “Request received OK” mapping node builds a simple text message that is returned to the client. This is done by building a JSON object message, and using an XPath “concat” statement to include the EMPNO element into the response message.  Close the map.

1.2.4 Explore the asynchronous REST operation

1.	In the HR_Service Resources folder, expand /**employees**/**complete**/**backend**, and look at the **addNewEmployeeIntoDB** operation. 
2.	As above, scroll to the right to open the subflow implementation of this operation. 

3. Open the map “Insert new Employee into database”.



This map has an input assembly in JSON format, which contains a full EmployeeAddUpdateCompleteRequest assembly. Elements from this input message are used in the various database functions in this map.

The map performs three database functions:

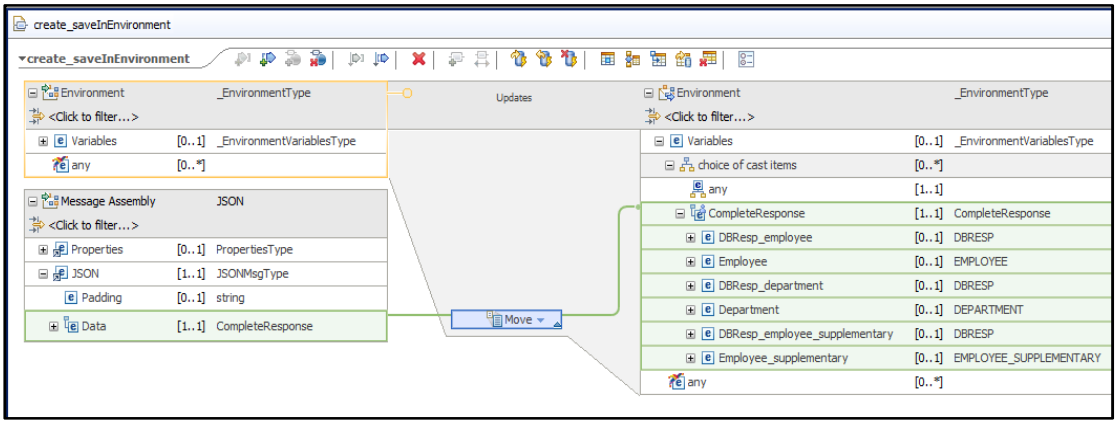
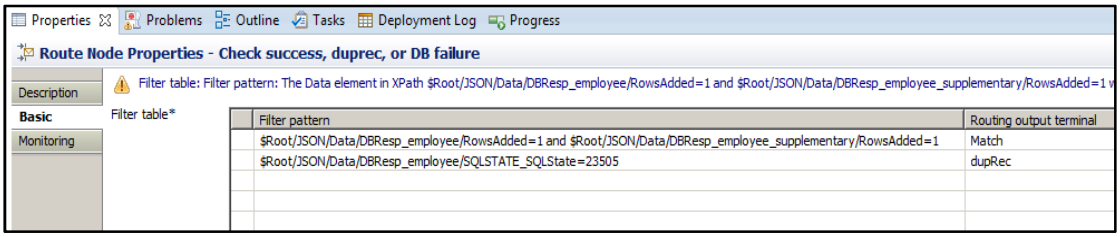
- Retrieves the DEPARTMENT row for the requested EMPNO from the DEPARTMENT table.
- Inserts the EMPLOYEE data into the HRDB/EMPLOYEE table (using the DEPARTMENT details just retrieved).
- Inserts the EmployeeSupplementary data into the EMPLOYEE_SUPPLEMENTARY table.

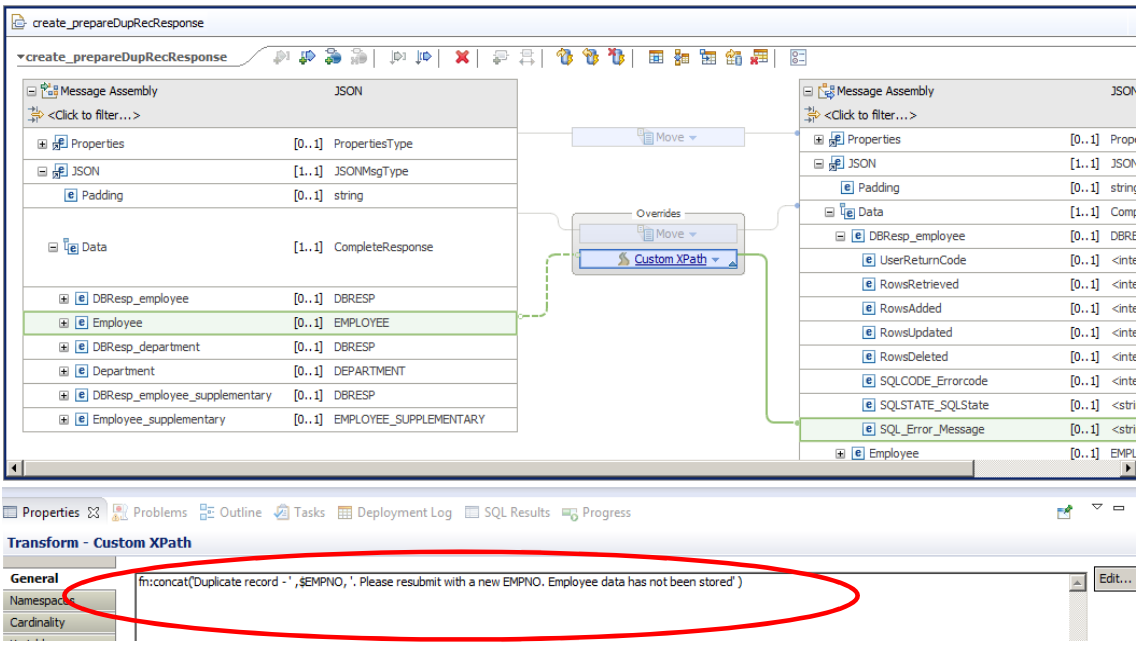
In the “Return” transforms of each insert, the user return code, and number of rows added, is set on the appropriate output message assembly. This information is used for routing later in the flow.

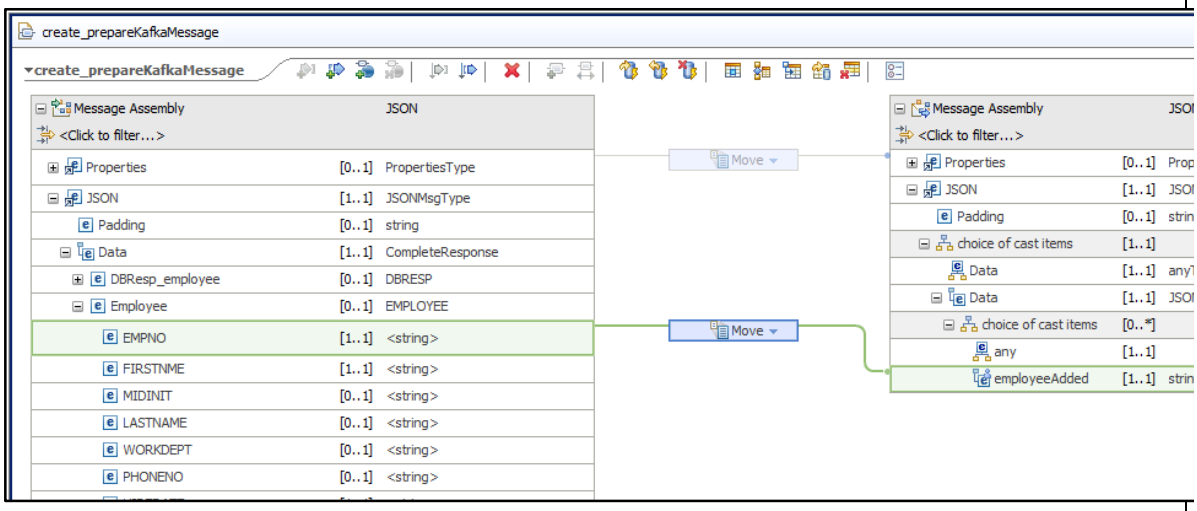
Failure scenarios such as “duplicate record” are handled by saving database returned data such as SQLSTATE in the output message.

The Failure transform of the EMPLOYEE insert is also configured to pass the EMPNO to the output assembly. This is to enable a database failure to be reported properly, but note that the full employee data is not recorded in the event of failure (eg. a duplicate record). An appropriate mechanism would normally be provided for this situation.

Close the map.

4.	Open the saveInEnvironment mapping node. The map performs a simple copy (Move) of the CompleteResponse input to the Environment tree.  Close the map.
5.	Highlight the Route node (Check success, duprec, or DB failure) and review the node properties. • The Match terminal is used when the database insert was successful (when the number of rows added was not zero for **both** tables). • The dupRec terminal is used when the SQLSTATE value is 23505 (SQL duplicate row) for the EMPLOYEE table only. You can extend the flow yourself if you want to check for other specific returns. You will see a message suggesting that the Data element in the Filter pattern was not found in the XML schema. This is because the XPath Expression builder does not support the JSON form of messages, so XPath evaluation expressions (including filter patterns) must be manually built. 

6.	Open the createDupRecResponse map node. Review the various mappings that are provided. In particular, the SQL_Error_Message is set with a Custom XPath transform, setting the message to a more readable form of the SQL error message. Note that the input EMPLOYEE/EMPNO element is connected to the Custom XPath transform, and referenced in the fn:concat Xpath statement. Close the map. 

7.	Open the **prepareKafkaMessage** mapping node. Only one element, EMPNO, is actually published to Kafka, so the output assembly was created dynamically by adding the **employeeAdded** element, using the "Add-User Defined" context menu item in the map editor.  Close the map.

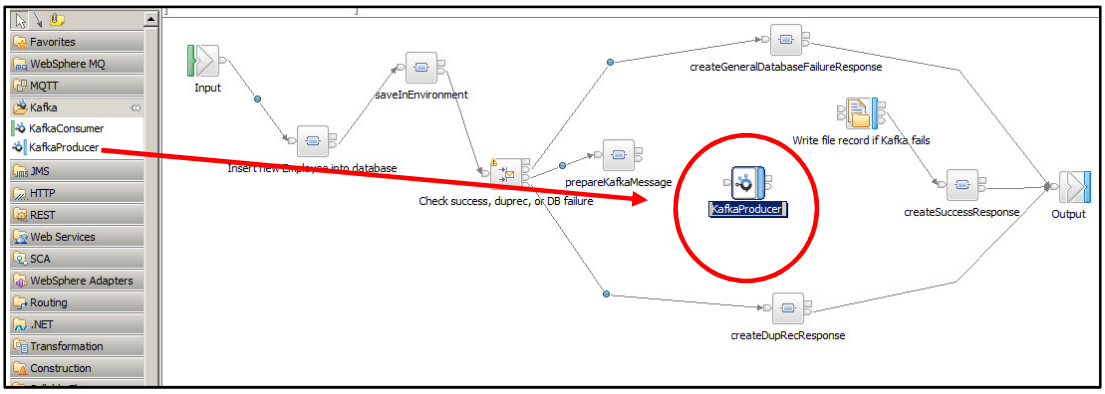
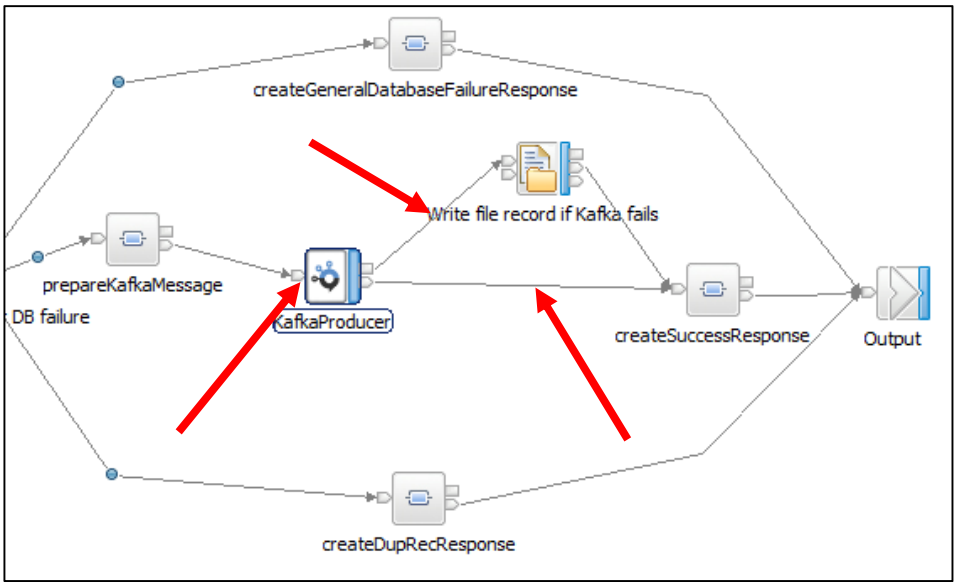
1.3 Extend the HR_Service implementation

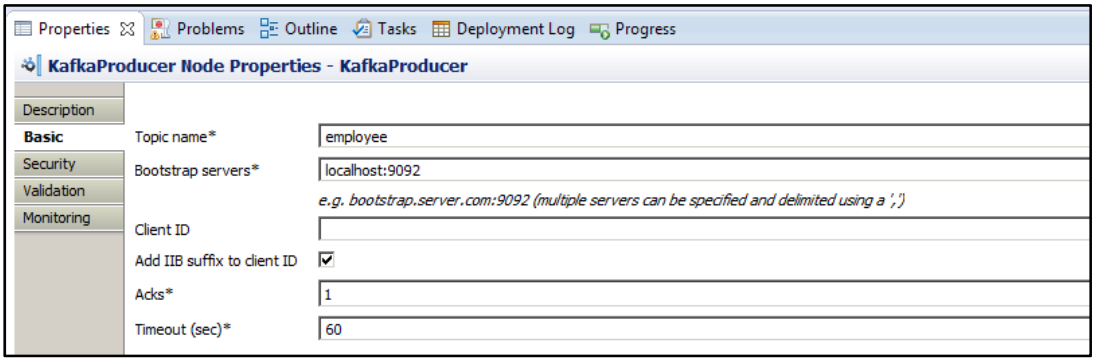
In this section, you will complete the implementation of the various operations and flows.

You will

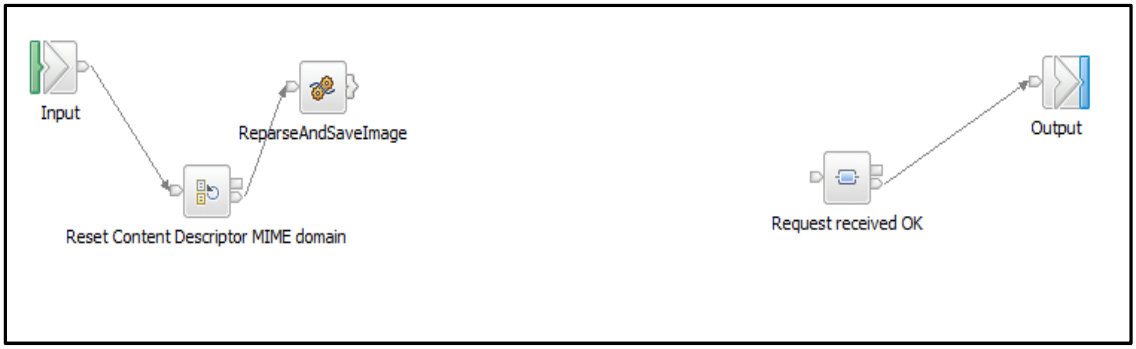
- Add a KafkaProducer node to the REST API operation addNewEmployeeIntoDB.
- Add a RESTAsyncRequest node to the createEmployeeFromMultipart operation.
- Complete the implementation of the asynchronous REST operation by adding a corresponding RESTAsyncResponse node in a separate application.
- Deploy and test the completed applications.

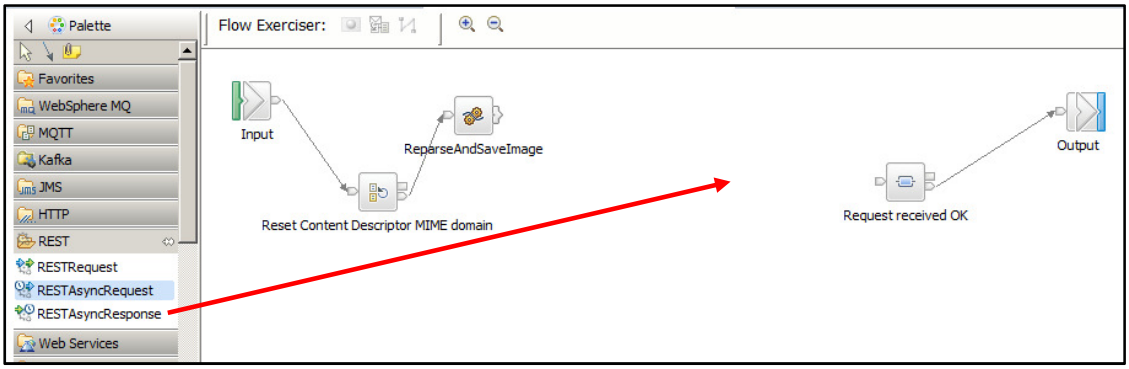
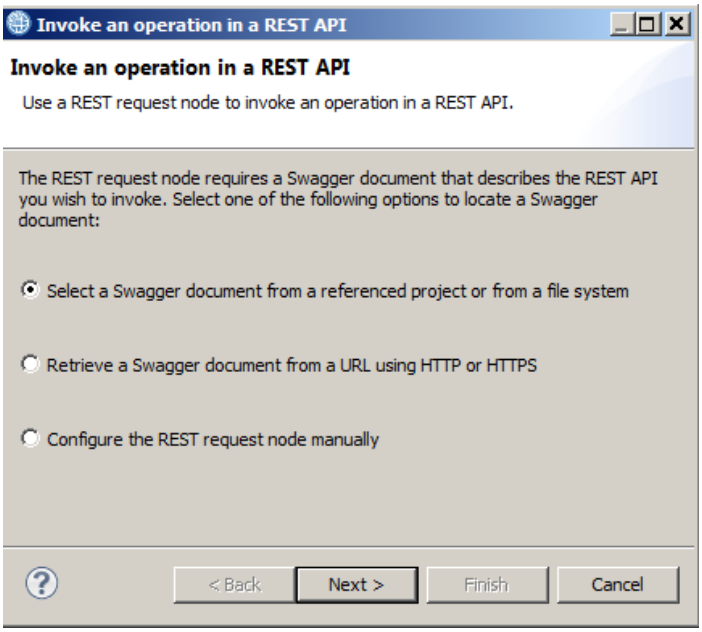
1.3.1 Add the KafkaProducer node to the addNewEmployeeIntoDB operation

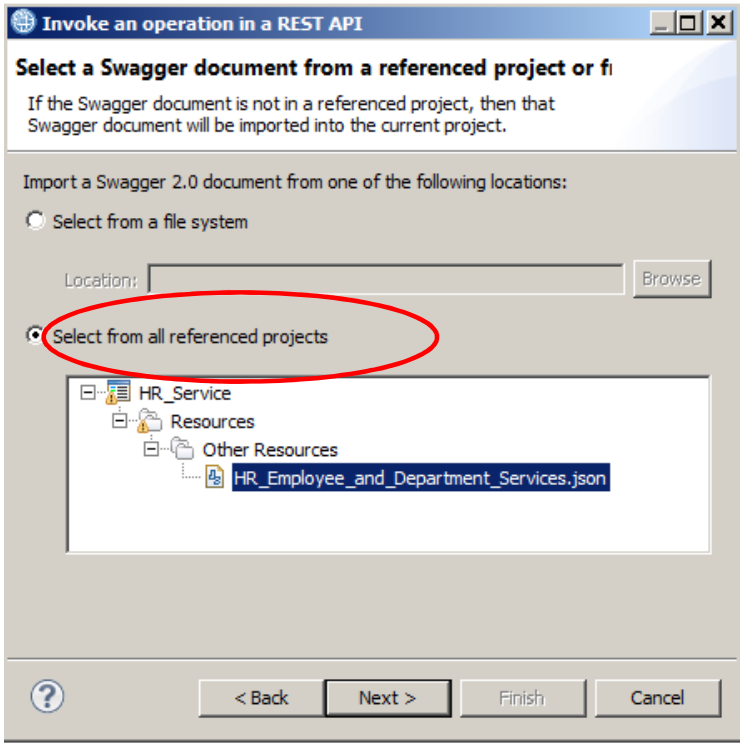
1.	Return to the **addNewEmployeeIntoDB** subflow. From the Kafka folder in the node palette, drop a KafkaProducer node onto the flow, in the open area as shown. 
2.	Add three connectors to the KafkaProducer node, as shown below. When connecting to the input of the FileOutput node, make sure you select the “In” terminal. 

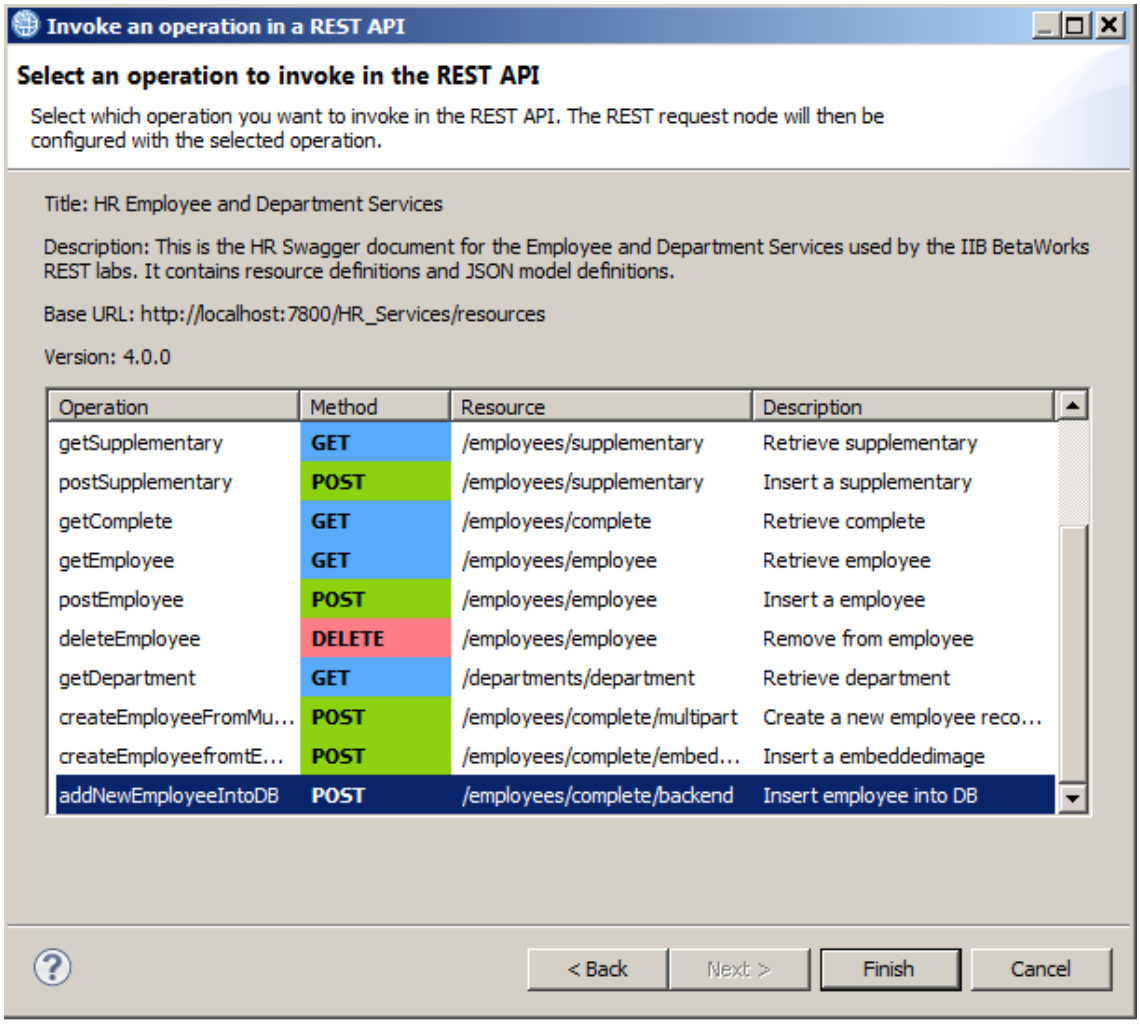
3.	Select the KafkaProducer node to highlight the node properties. Set the following properties: • Topic name : employee • Bootstrap servers: localhost:9092 • Acks: 1  Save the subflow (Ctrl-S).

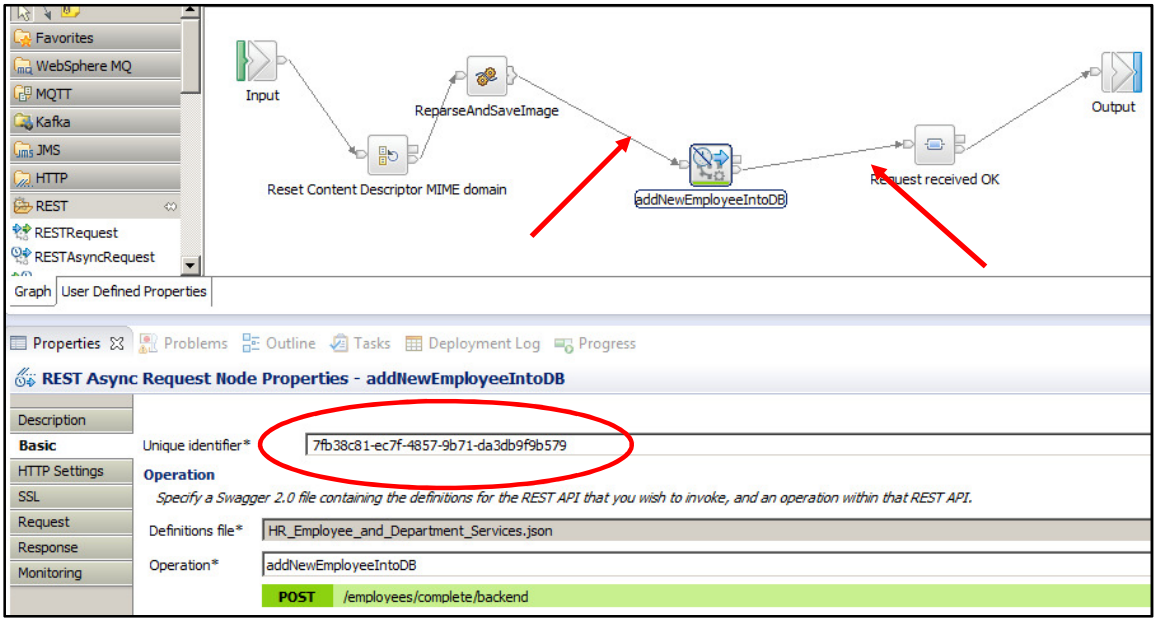
1.3.2 Add a RESTAsyncRequest node to the createEmployeeFromMultipart subflow

1.	Switch to the createEmployeeFromMultipart subflow. 

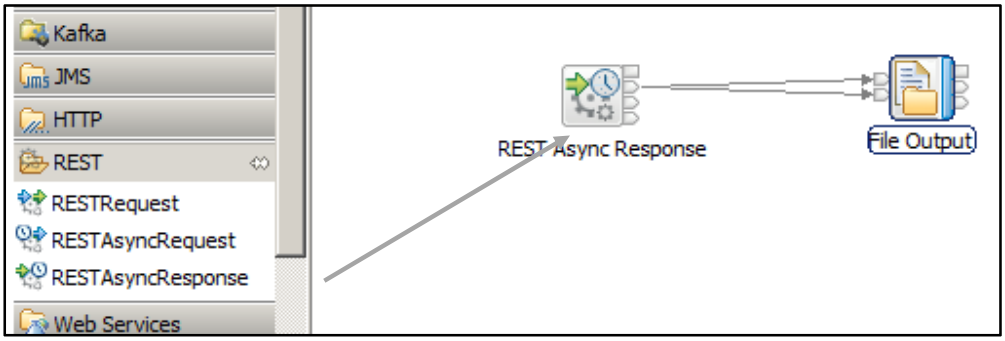
2.	From the node palette, in the REST folder, drop a RESTAsyncRequest node onto the flow editor, in the position as shown. 
3.	When the node is dropped on to the editor, a new window will open. The new node can be configured using information from a variety of sources. In this lab, you will use the Swagger document that represents the local HR_Service REST API, so leave the default selection and click Next. 

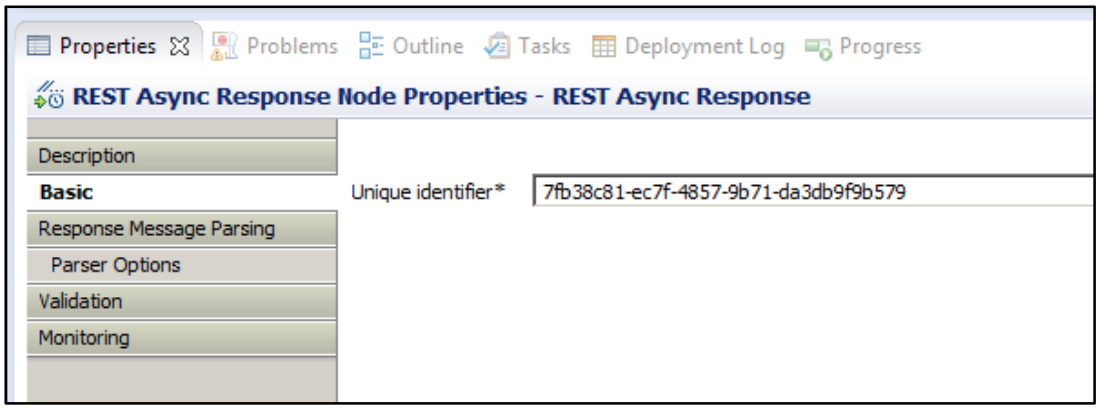
4.	At the next window, you can choose the location of the Swagger document. Choose the “Select from all referenced projects”, and highlight HR_Employee_and_Department_Services.json. Click Next. 

5.	At the next window, select the required REST operation. The addNewEmployeeIntoDB operation should be at the bottom of the list, so scroll down and select this operation. Click Finish. 

6.	The new node will have been added to the flow. Connect the node as shown. Select the node, and review the node properties. No properties need to be changed, but note in particular the “Unique identifier”, which has been set to a value generated by the IIB Toolkit. You can specify your own value here, but in this case, accept the generated value.  Save the subflow (Ctrl-S).

1.3.3 Add a RESTAsyncResponse node to the receiving application

1.	Finally, complete the application that will handle the RESTAsyncResponse. Expand the HR_Service_AsyncBackend application, and open the HR_Service_AsyncResponse_insertDB message flow. At the moment, this just has one node, a FileOutput node, which is used to record database updates. 
2.	From the node palette, drop a RESTAsyncResponse node onto the flow, and connect the Out terminal to **both** the In terminal and the Finish terminal of the FileOutput node. 

3.	Highlight the RESTAsyncResponse node, and view the Properties of the node. No changes need to be made, because the IIB Toolkit has automatically generated the same Unique identifier for the response node. As above, this can be changed to a value of your own choosing, and if the node was located in a different workspace, then this would need to be manually set. In this case, leave the value unchanged (assuming you made no changes to the corresponding RESTAsyncRequest node).  Save the message flow.

1.4 Explore and start the Kafka servers

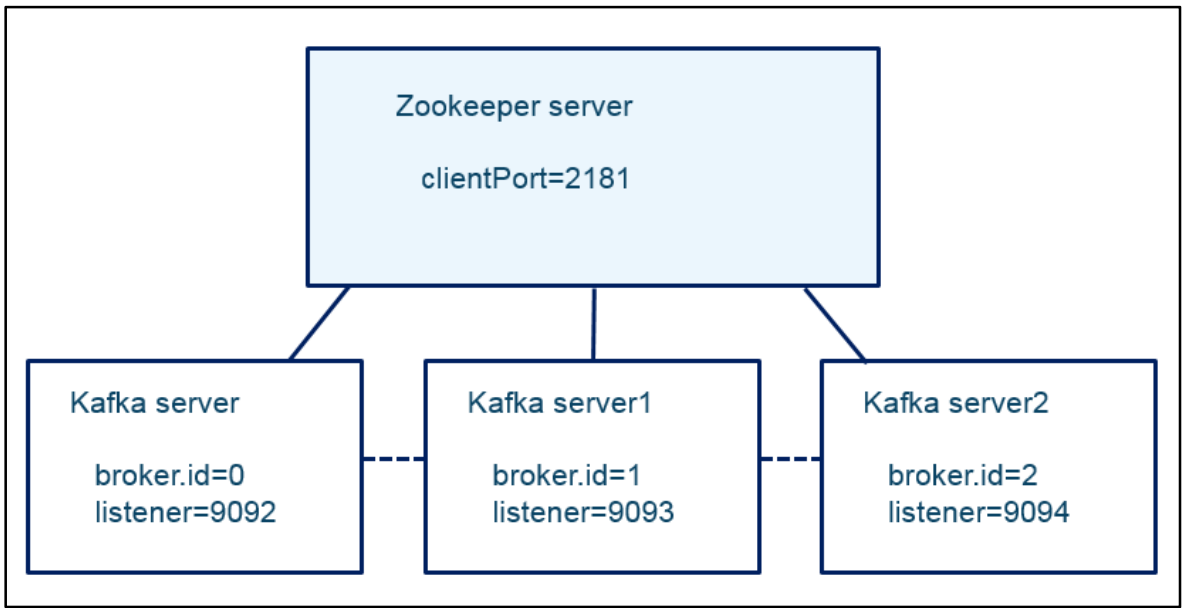
The supplied VM system that is provided for this lab is supplied with a local installation of the Apache Kafka system.

1.4.1 Kafka configuration for IIB workshop

Kafka is installed in **c:\tools\kafka_2.11-0.10.1.0**. In the **\bin\windows** folder, there are a number of ".bat" files that control various aspects of the Kafka system. For ease of use, some of these have been copied into the folder **c:\student10\kafka\commands**.

On this system, Kafka has been configured to use a single Zookeeper server and three Kafka servers. This enables a topic Replication Factor of three.

The Kafka servers are shown schematically below. Note that all the servers are defined locally, so all have a unique listener port.



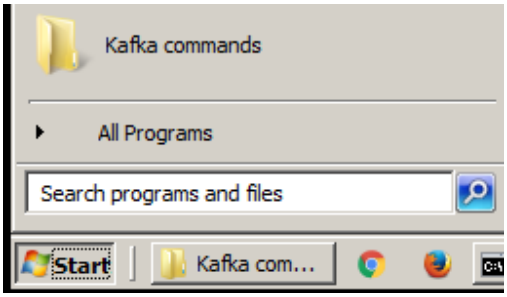
1.4.2 Explore the Kafka Configuration

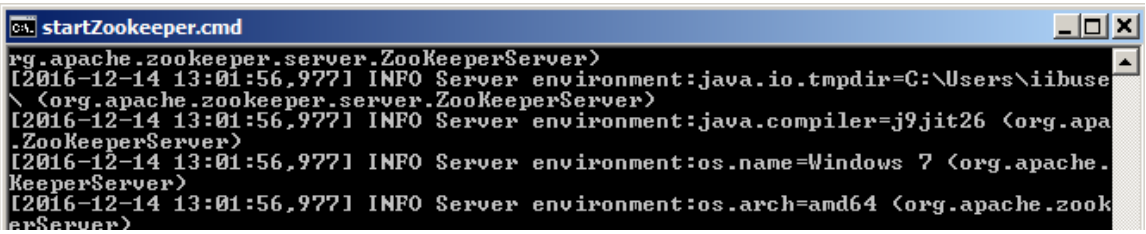
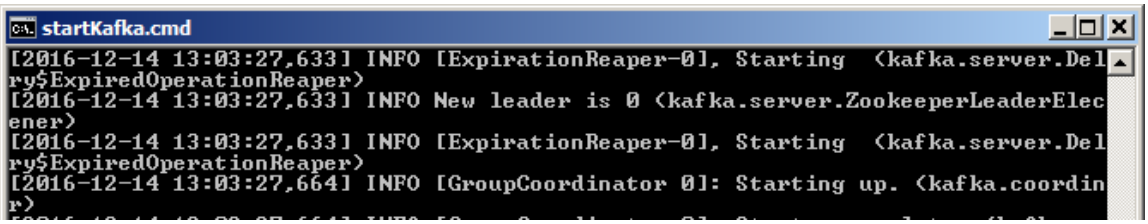
1.	In Windows Explorer, navigate to the folder **c:\student10\kafka\config**.
2.	Open the file **zookeeper.properties** (right-click and open with Notepad++). Review the properties, but do not make any changes. #dataDir=/tmp/zookeeper dataDir=c://kafka/zookeeper # the port at which the clients will connect clientPort=2181 # disable the per-ip limit on the number of connections since this is a non-production config maxClientCnxns=0 Close the file when complete.
3.	Open the file server.properties. Most properties have been left at the default values. The following properties have been set as follows: • Delete.topic.enable=true (allows topics to be removed at server restart) • Broker.id=0 (unique number for each Kafka server) • Listeners=PLAINTEXT://:9092 (unique port for each Kafka server) • Log.dirs=c:/kafka/kafka-logs (location of kafka log files) # Topic deletion properties delete.topic.enable=true ################### Server Basics ################### # The id of the broker. This must be set to a unique integer for each broker. broker.id=0 ############### Socket Server Settings ############### listeners=PLAINTEXT://:9092 ################## Log Basics ################ # A comma separated list of directories under which to store log files log.dirs=c:/kafka/kafka-logs Close the file when complete.

4.	The server-1.properties and server-2.properties are configured similarly, as follows: server-1.properties • Delete.topic.enable=true • Broker.id=1 • Listeners=PLAINTEXT://:9093 • Log.dirs=c:/kafka/kafka-logs-1 server-2.properties • Delete.topic.enable=true • Broker.id=2 • Listeners=PLAINTEXT://:9094 • Log.dirs=c:/kafka/kafka-logs-2

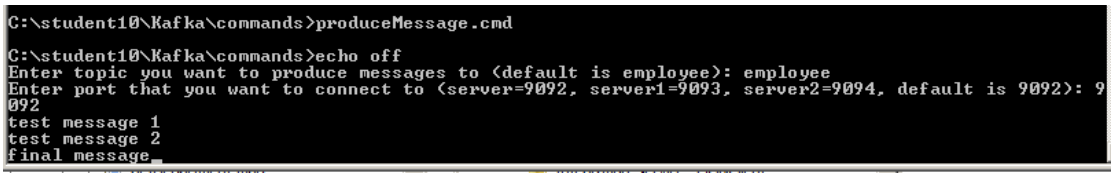
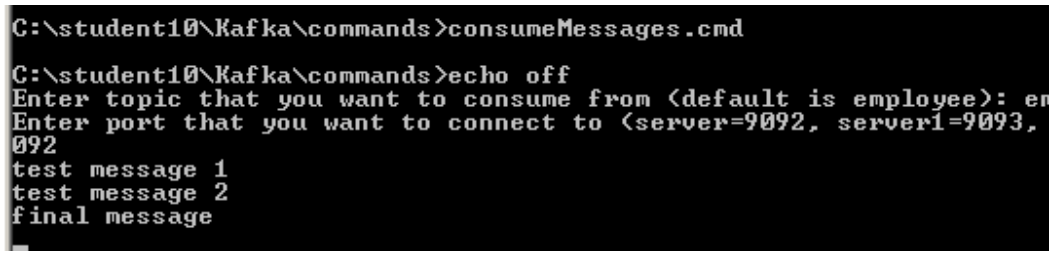
1.4.3 Start the Kafka servers

Windows shortcuts have been provided for the Kafka commands that are required to start the various servers.

1.	From the Windows Start menu (or from the desktop), open the folder “Kafka commands)  The following shortcuts will be available: 

2.	Open (run) **startZookeeper.cmd**. A Windows DOS command window will open and the zookeeper server will be started. A significant amount of log output will be produced. When started this way, the “startZookeeper.cmd” name will be shown in the title line of the DOS window. 
3.	Open (run) startKafka.cmd. As above, the server will produce some log output. 
4.	Repeat with **startKafka-server1.cmd** and **startKafka-server2.cmd**.
5.	At this point, all Kafka servers are running, so now create a new topic. Open a new DOS window, and change directory to **`c:\student10\kafka\commands`**

6.	Run the **createTopic.cmd** file. Provide the following values: • Topic: employee • Replication factor: 3 • Partitions 2 `C:\student10\Kafka\commands>createTopic.cmd` `C:\student10\Kafka\commands>echo off` `Enter topic to create (default is employee): employee` `Enter Replication Factor for employee (default is 1): 3` `Enter number of partitions to create (default is 1): 2` `Created topic "employee".`
7.	Run the command file **`listTopics.cmd`**. The command will return “employee”. `C:\student10\Kafka\commands>cmd /c "kafka-topics.bat --list --zookeeper localhost:2181"` `employee`
8.	Run the command file **`describeTopic.cmd`**. Provide “**employee**” as the topic name. The command will return information about the replication factor and partitions of the “employee” topic. If you have followed the instructions above, you will see output similar to that below. `C:\student10\Kafka\commands>describeTopic.cmd` `C:\student10\Kafka\commands>echo off` `Enter topic to describe (default is employee): employee` `Topic:employee  PartitionCount:2  ReplicationFactor:3  Configs:` `  Topic: employee Partition: 0  Leader: 2  Replicas: 2,0,1 Isr: 2,0,1` `  Topic: employee Partition: 1  Leader: 0  Replicas: 0,1,2 Isr: 0,1,2`
9.	Run the command **`consumeMessages.cmd`**. Specify the “employee” topic, and connect to the Kafka server with port 9092. `C:\student10\Kafka\commands>consumeMessages.cmd` `C:\student10\Kafka\commands>echo off` `Enter topic that you want to consume from (default is employee): employee` `Enter port that you want to connect to (server=9092, server1=9093, server2=9094, default is 9092): 9` `092`

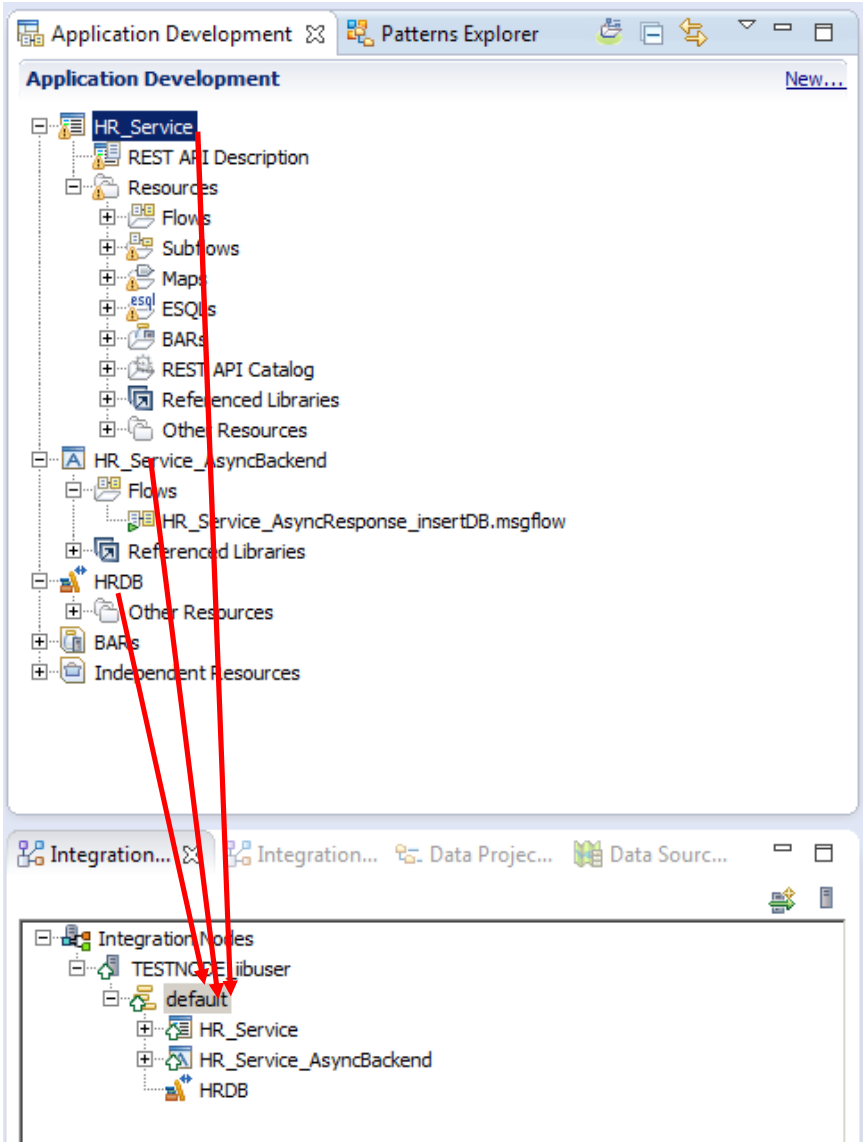
10.	Open a further DOS window, and navigate to c:\student10\kafka\commands. Run the command **`produceMessage.cmd`**. Specify the “employee” topic, and connect to Kafka with port 9092. Type some text message input, as shown below. Each message is terminated with the Return key. 
11.	Back in the consumeMessages window, observe that the text messages you just produced have been consumed by the consumeMessages client application.  You have now verified that the local Kafka system is configured correctly, and can be used by the IIB applications.

1.5 Test the REST API

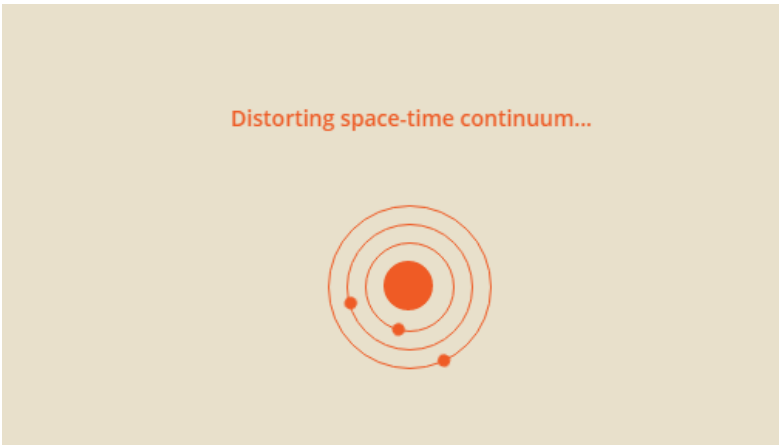
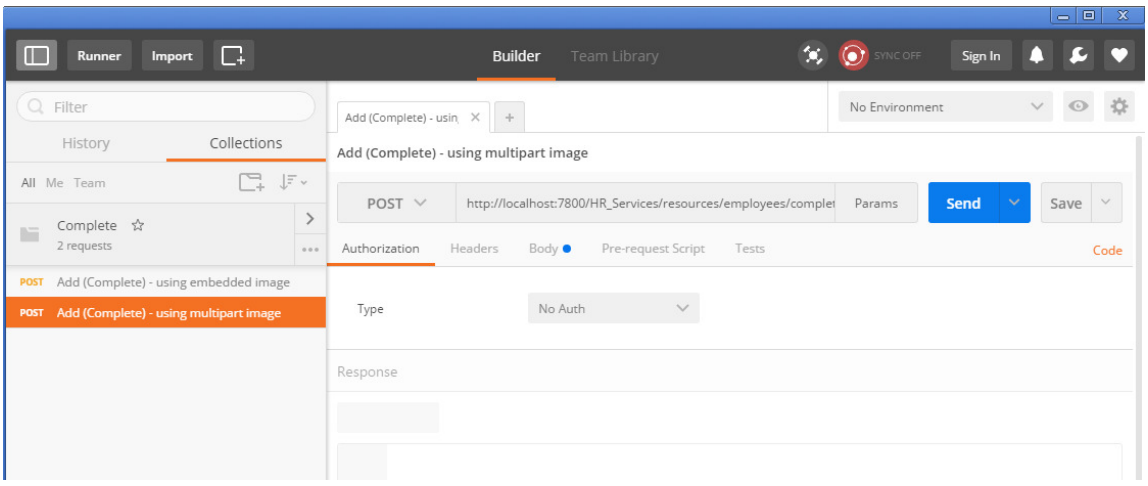
1.5.1 Deploy HR_Service and the HRDB shared library

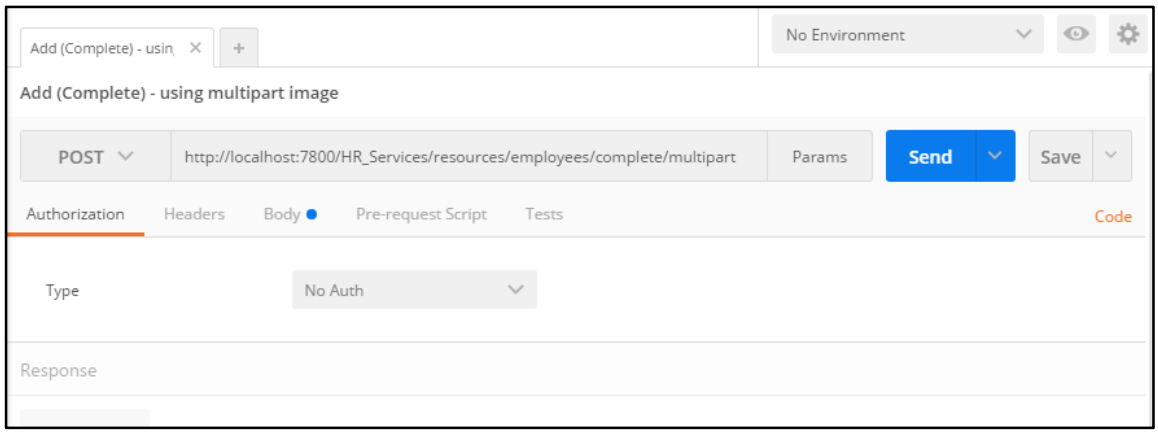
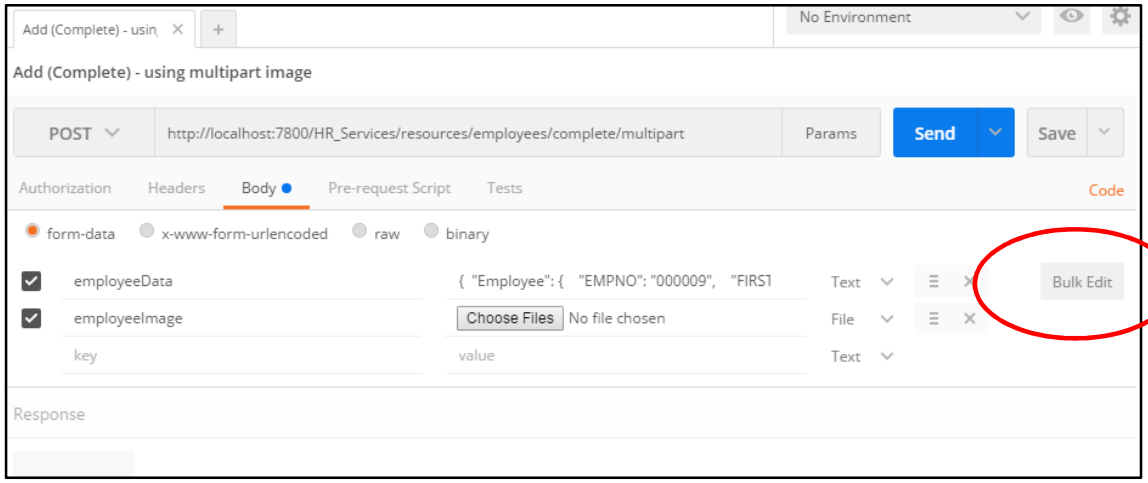
1. Deploy the following resources:
   - HRDB shared library
   - HR_Service
   - HR_Service_AsyncBackend

   Note – the HR_Service_AsyncBackend must be deployed at this time. This is because IIB will check for the existence of the REST asynchronous unique identifier when the RESTAsyncRequest node is executed.



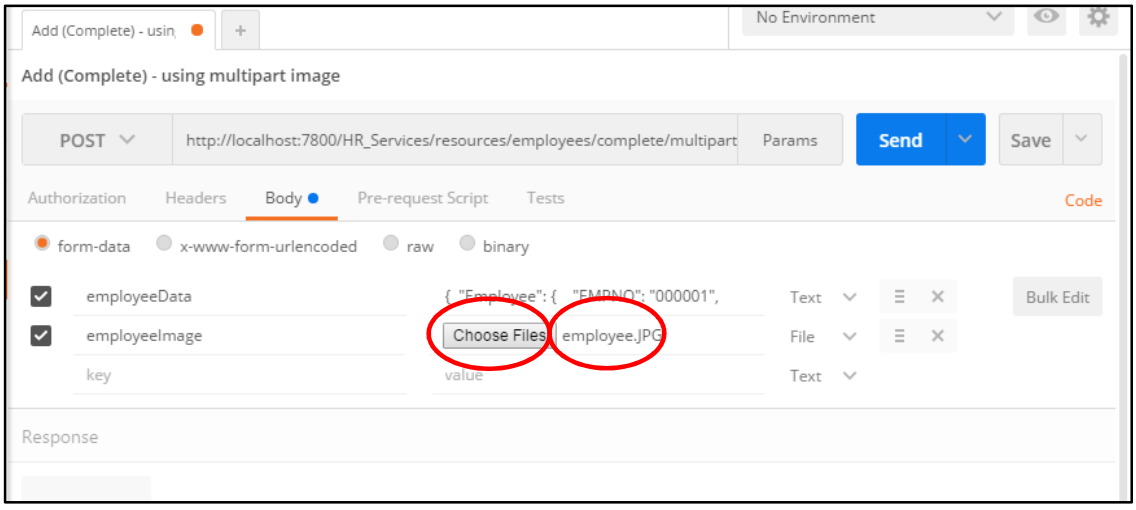
1.5.2 Test HR_Service

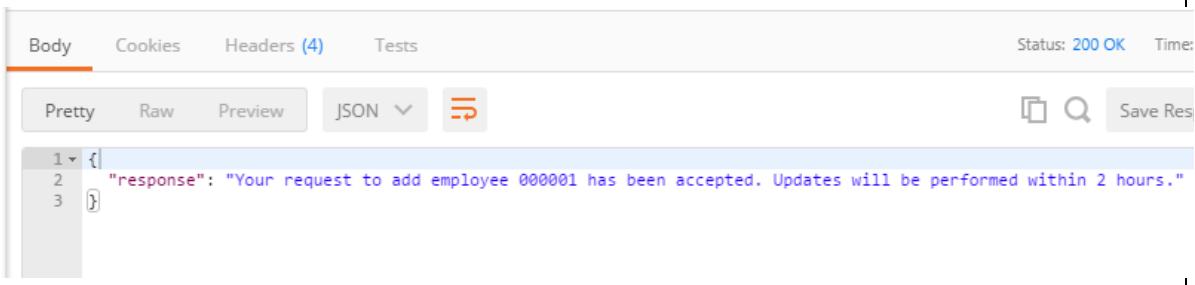
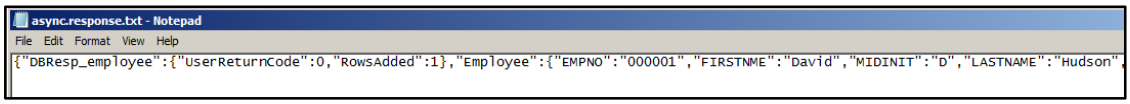
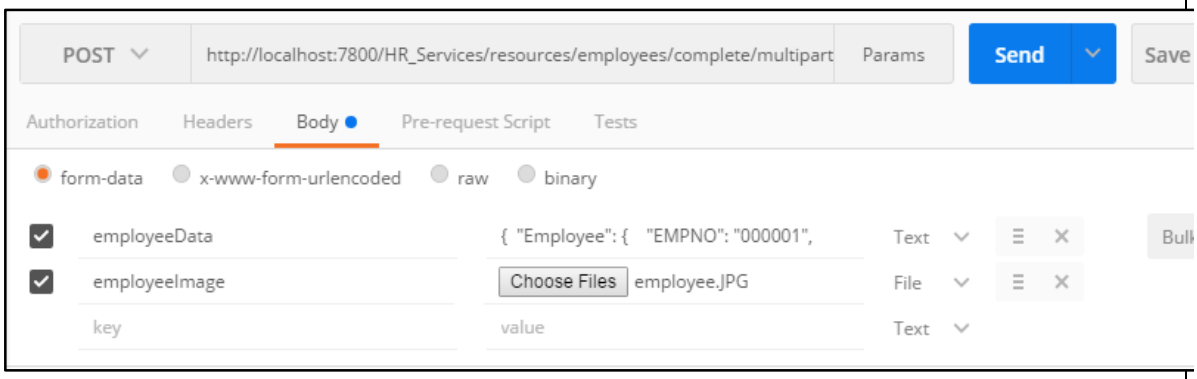
1.	From the Start menu, start the Postman tool (type Postman into the Start Search menu). After the progress message…  … you will see the Postman main menu.
2.	The required Postman project should already be available. You may need to click on the ‘Complete’ tab so they are shown. Highlight the second POST request, Add(Complete) using multipart image. 

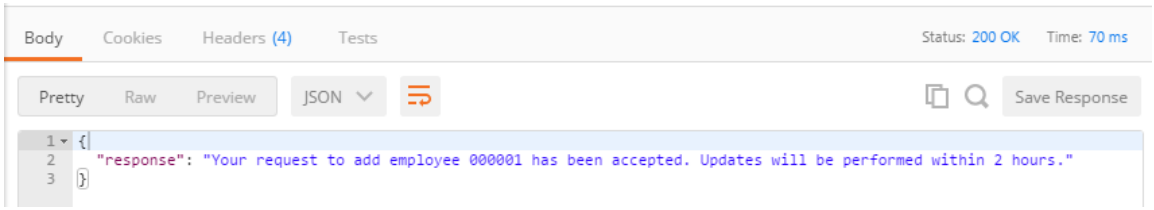
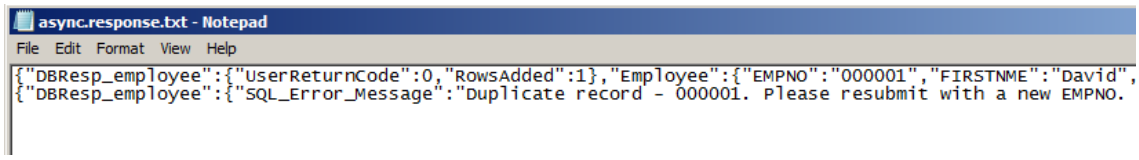
3.	On the right pane, note that the URL is set to the required URL for the HR_Service operation. 
4.	Click the “Body” tab. Note the format of the message is “form-data”. Selecting this option means that you can construct the payload of the message using a multipart format. Note that the message has two parts: • employeeData – the JSON part of the multipart message, with a message payload representing a new EMPLOYEE. • employeeImage – the binary part of the multipart message. In this example, you will attach a jpg image of the new employee. Note that the names employeeData and employeeImage do not need to match any part of the message elements sent to the REST API. 

5.	Click “Bulk Edit” (above) to see the input message JSON data in its entirety. Click Key-Value-Edit to return to the earlier display. 

1.5.3 Test with Postman

1.	Specify the name of the employeeImage file. Using the Choose Files button, set this to **`c:\student10\REST_Request_Async_IC17\data\employee.jpg`** Set the employee number (EMPNO) to one that is known not to already exist (eg. 000001). Click Send. 

2.	An immediate response will be received, as shown below. 
3.	Switch to the DOS window that has subscribed to the “employee” topic. Note that employee 000001 has been published to the Kafka server. 
4.	In Windows Explorer, open the file **c:\student10\REST_Request_Async_IC17\output\async_response.txt** Note that the employee update has been recorded in the text output file. Close the file. 
5.	In Postman, click Send again (without changing the employee number). 

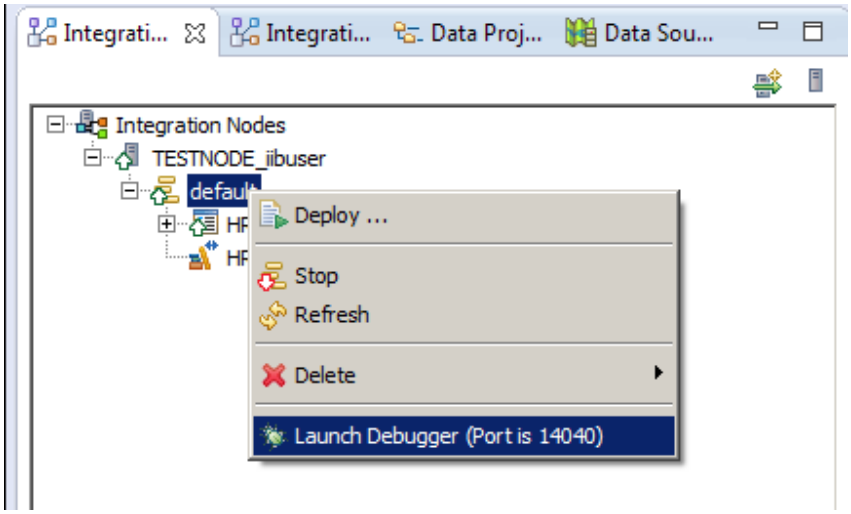
6.	You will again see the successful response message: 
7.	Reopen the file **async.response.txt**. Note that a new line has been added that indicates that EMPNO=000001 was a duplicate record, and has not been added to the database. In the integration as designed, for a duplicate row, the input data was not recorded elsewhere. It would be possible to do this, and include a retry option with a different employee number. This is left as an exercise for the reader. Close the file. 

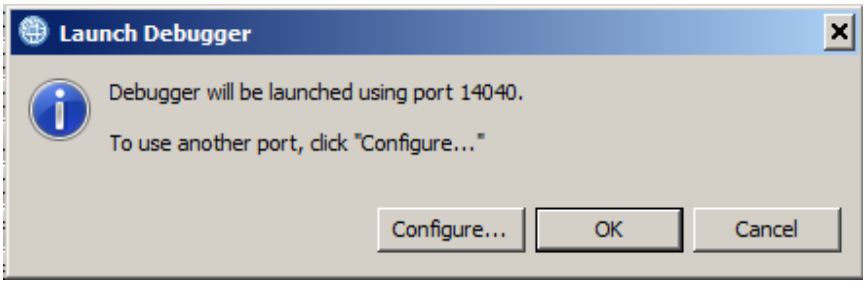
1.6 Investigate in more detail using debug mode (optional extension)

In this section, you will perform the test again, whilst having the IIB REST API in debug mode. Using this tool, you will see the message tree at various stages in the flow, and see the multipart, MIME and JSON messages as they are manipulated by the IIB flow.

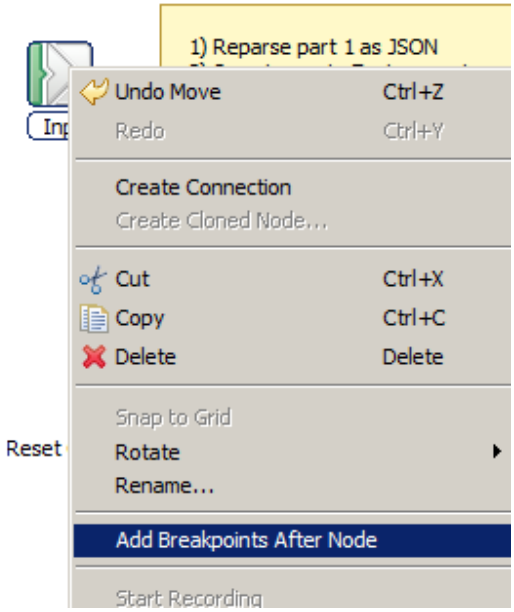
1. First, activate the debugger on your IIB node/server.

   Right-click the IIB server, and select Launch Debugger. If you have not already configured a port for the debugger to use, use the configure button to specify a suitable port.

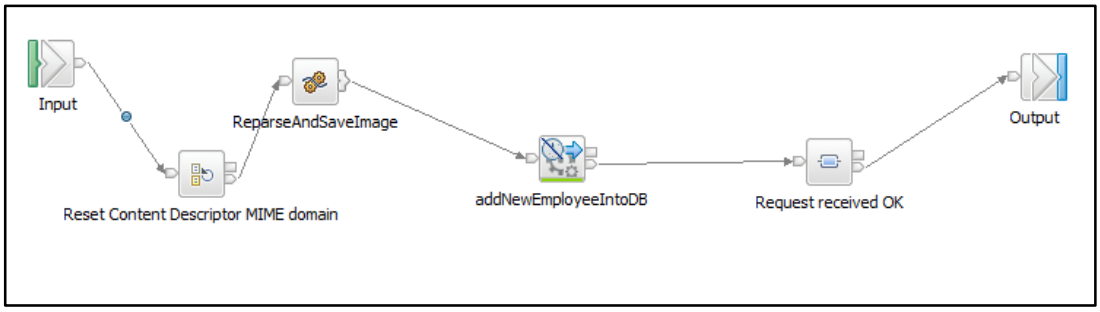




2. In the createEmployeeFromMultipart subflow (should still be open from above), add a breakpoint after the Input node (right-click, Add Breakpoints….".

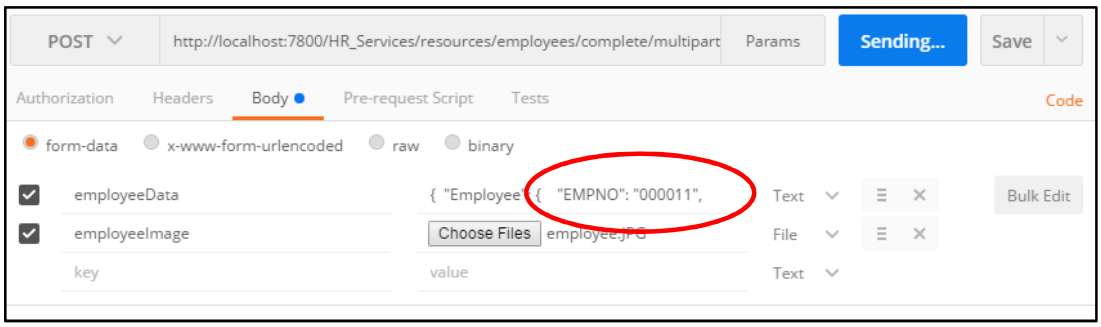


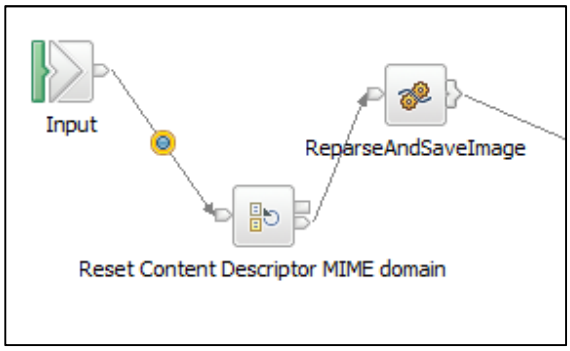
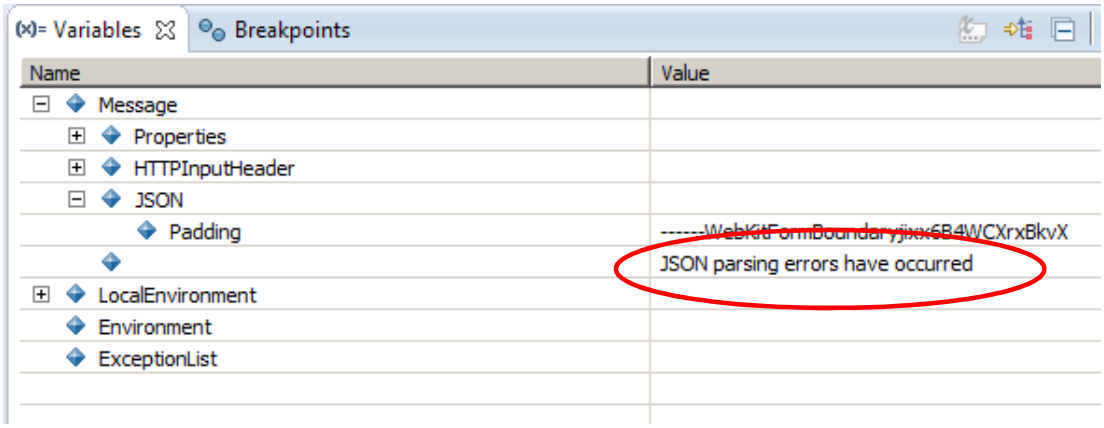
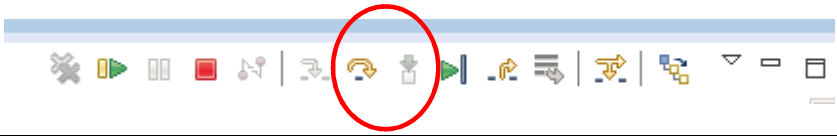
The flow will show the blue breakpoint on the connector.



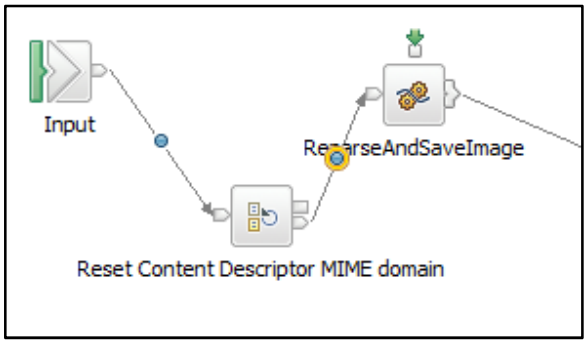
3. Invoke the test again. In Postman, provide a new value for EMPNO (eg. 000011), and click Send.

You may need to use the slide bar to move to the top of the Postman window to see the input data.



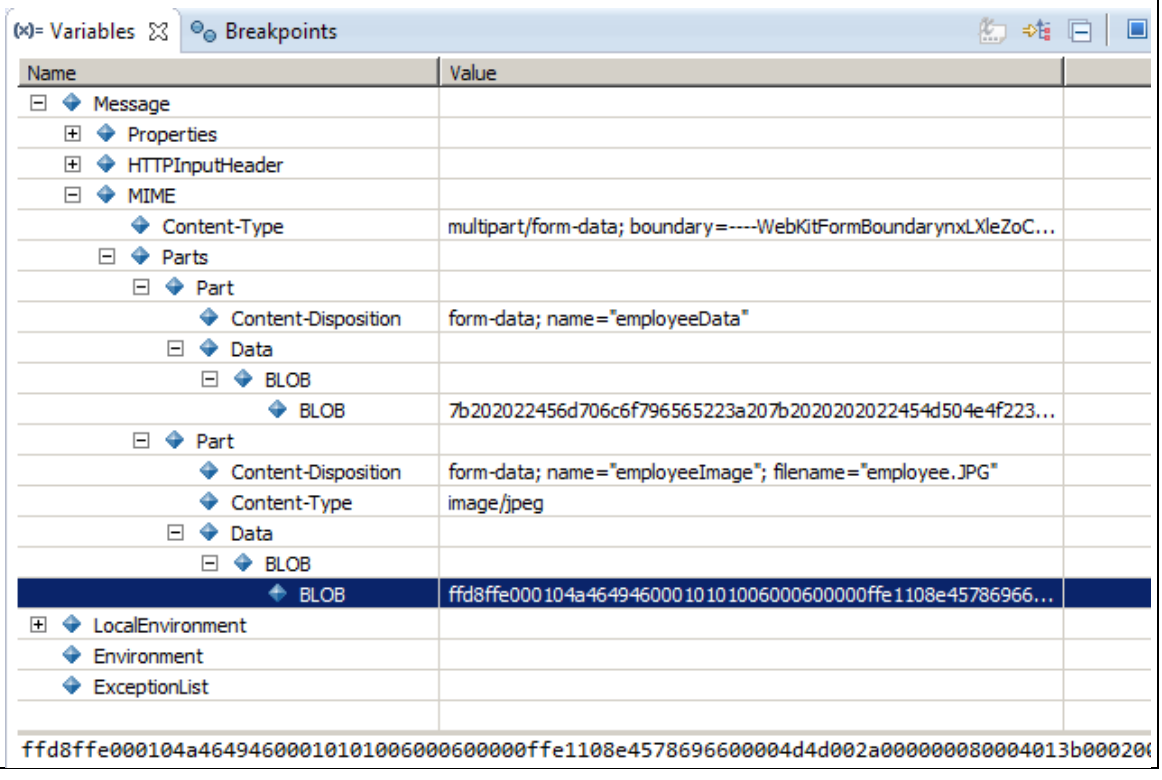
4.	The flow will start, and execution will stop at the first breakpoint. (Respond Yes to switch to the Debug perspective). 
5.	Highlight the debugger Variables view, and expand the incoming message. Note that no user data is visible, and JSON parsing errors have occurred. This is because the REST API is configured to expect JSON data, but the message payload is not JSON. It is a multipart message with a JSON component, and a binary component, so the message has failed to be parsed. 
6.	Click the Step Over icon in the debugger control view. 

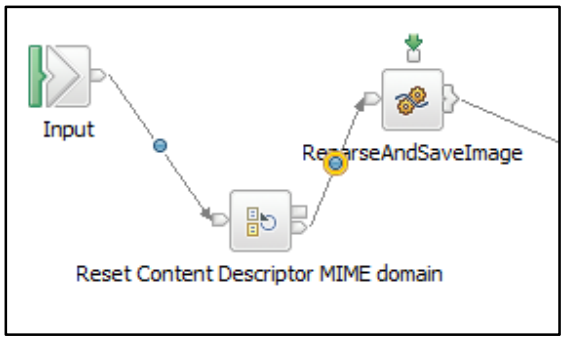
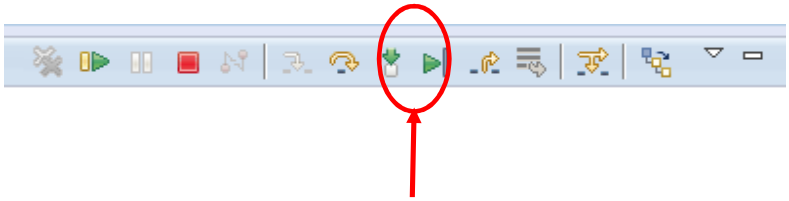
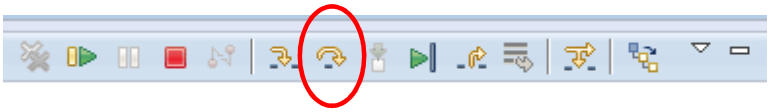
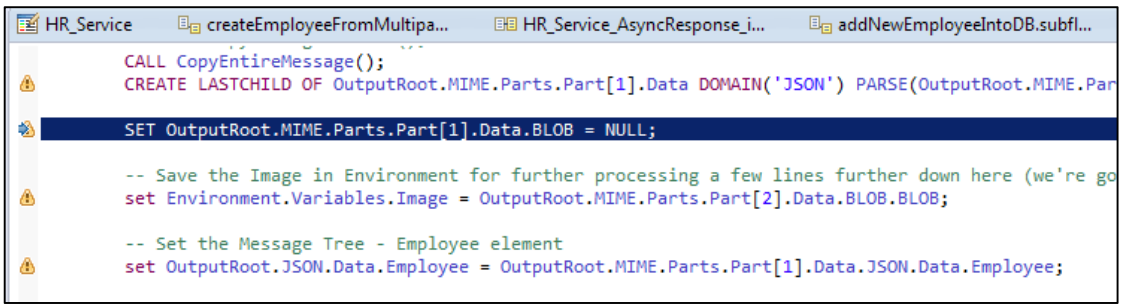
7. The debugger will pause after the Reset Content Descriptor node.



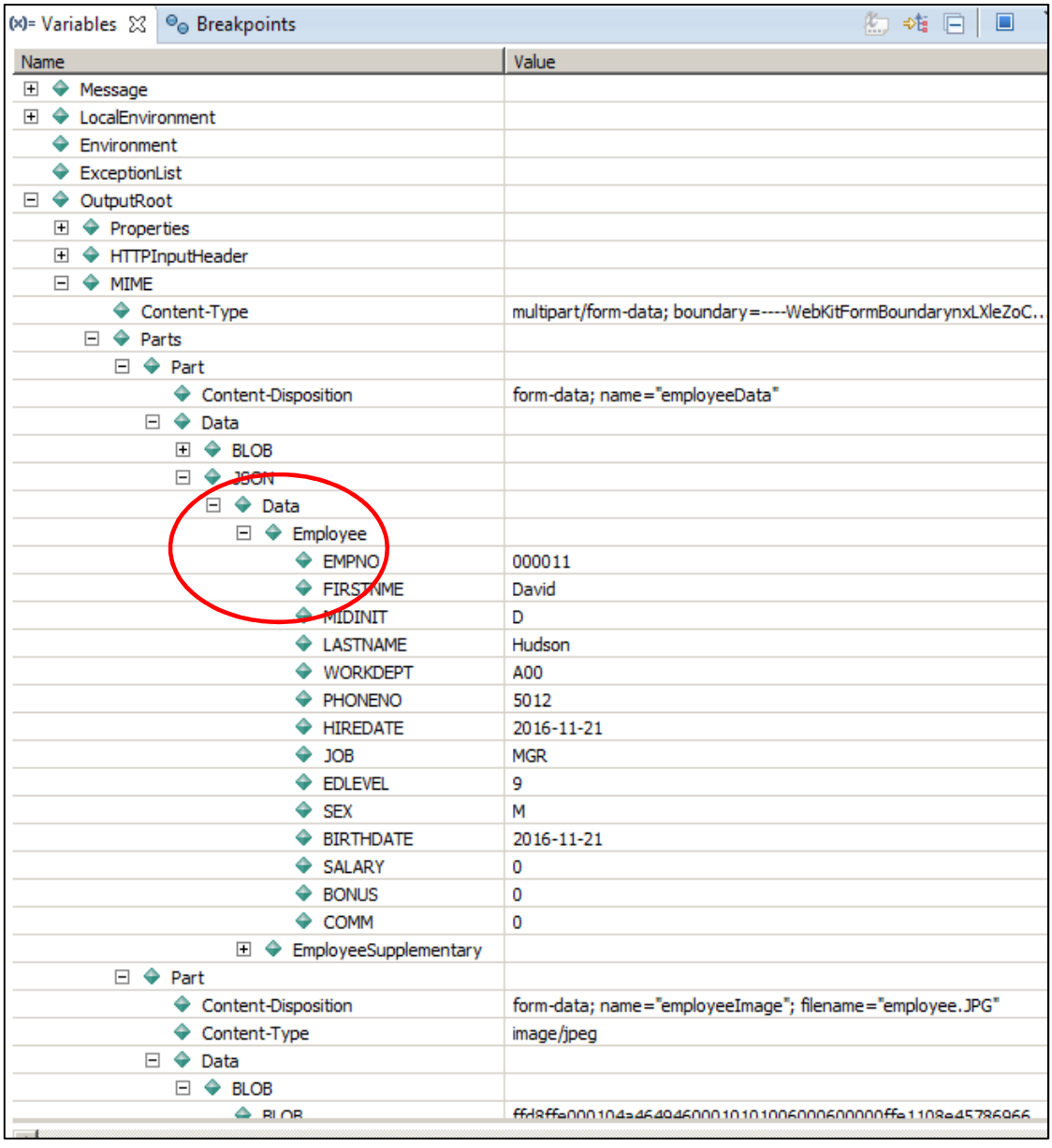
In the Variables view, expand the Message. Note that the message has been parsed by the MIME parser.
Fully expand the MIME message, and note that there are two parts to the message.

- The JSON part of the message. (The data has not yet been parsed by the JSON parser, so the data is a BLOB and is not yet readable. However, the Debug perspective in the Toolkit renders the data in a readable hexadecimal format, as shown below).
- Part 2 is the binary data, containing the attached jpg image. Note the Content-Disposition contains the filename, and the Content-Type contains the type of data. Selecting the BLOB item shows the raw data of the binary part of the message.



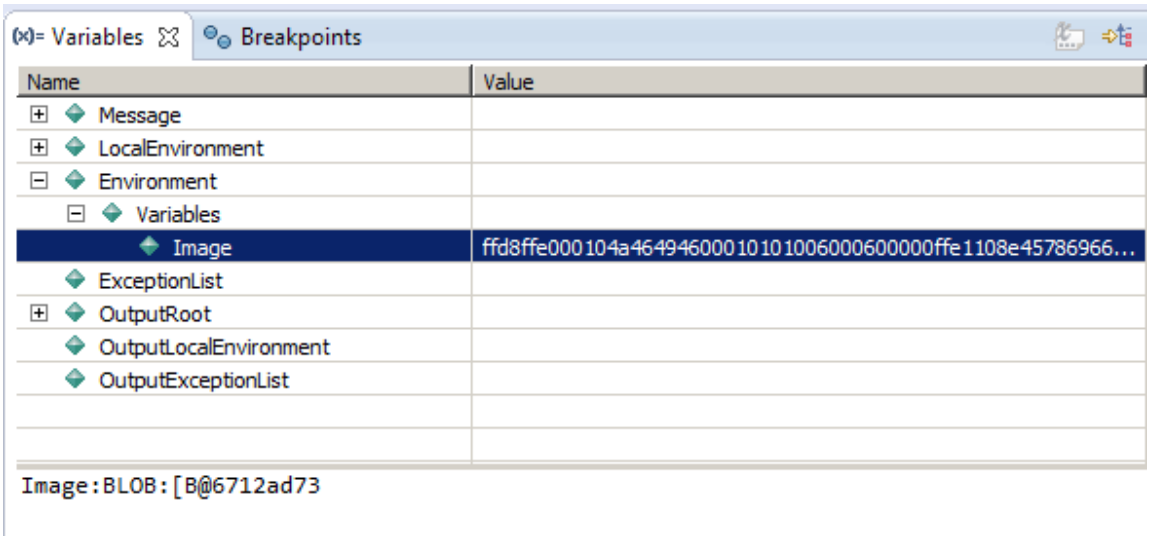
8.	The debugger is currently paused just before the ReparseAndSaveImage compute node. It is instructive to observe the ESQL, and how the message tree is manipulated, so click the green down-arrow, as shown on the debug control line, to enter the ESQL node in debug mode.  
9.	Click the Step Over icon a few times, until the **next** line to be executed (the highlighted line) is the line starting **`SET OutputRoot.MIME.Parts …`**  This means that the line starting **`CREATE LASTCHILD . .`** has just been executed.  The “Create LASTCHILD” statement does two things: 1) Creates a new element called “JSON” in the array OutputRoot.MIME.Parts.Part[1].Data 2) Parses the BLOB part of the message and uses this to populate the new element Parts.Part[1].Data.JSON. Because the BLOB data was a JSON message, this results in the EMPLOYEE message being recreated in the new output element. You will see this on the next page.

10. In the Variables view, expand OutputRoot. You will see that MIME section now has a new Data element under Parts.Part[1]. This Data element has been created in the JSON domain, so you are now able to see the Employee data in its fully parsed state (even though it is currently held under the MIME part of the message).



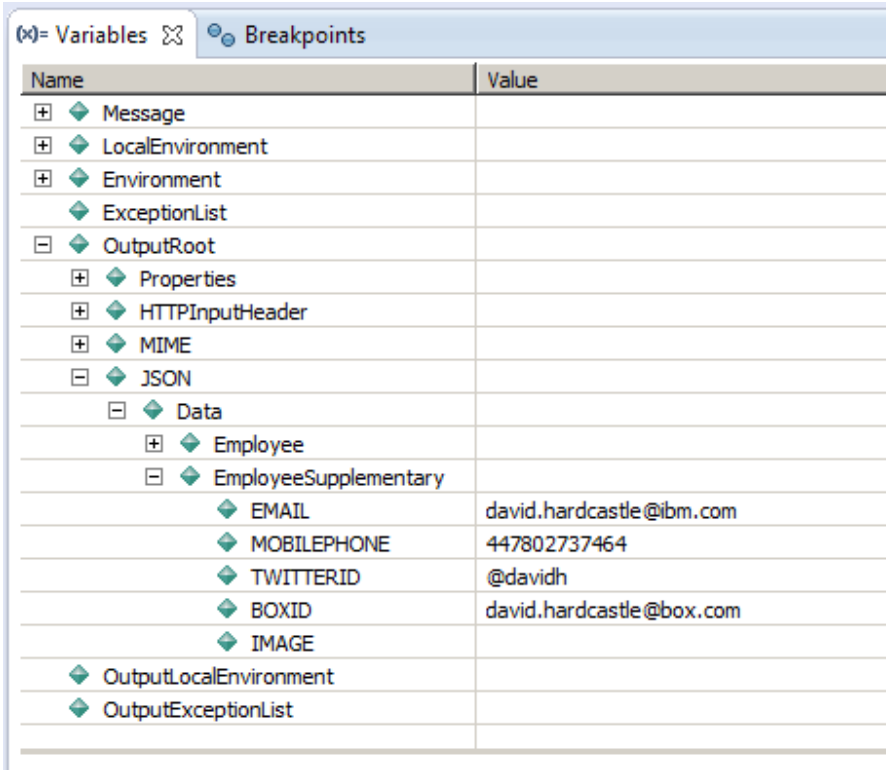
11.	In the debugger, step over once more. The line above the highlighted line shown below will have been executed. `CREATE LASTCHILD OF OutputRoot.MIME.Parts.Part[1].Data DOMAIN('JSON') PARSE(OutputRoot.MIME.Parts.` `SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;` `-- Save the Image in Environment for further processing a few lines further down here (we're going` `set Environment.Variables.Image = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;`
12.	In Variables, note that the OutputRoot, under the MIME section, does **not** now contain a BLOB folder (the last line just set it to Null). Variables / Breakpoints
13.	In the debugger, step over once more. `SET OutputRoot.MIME.Parts.Part[1].Data.BLOB = NULL;` `-- Save the Image in Environment for further processing a few lines further down here (we're` `set Environment.Variables.Image = OutputRoot.MIME.Parts.Part[2].Data.BLOB.BLOB;` `-- Set the Message Tree - Employee element` `set OutputRoot.JSON.Data.Employee = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.Employee;`

Name	Value
Message	
LocalEnvironment	
Environment	
ExceptionList	
OutputRoot	
Properties	
HTTPInputHeader	
MIME	
Content-Type	multipart/form-data; boundary=----WebKitFormBoundarynxLXleZoC.
Parts	
Part	
Content-Disposition	form-data; name="employeeData"
Data	
JSON	
Part	
Content-Disposition	form-data; name="employeeImage"; filename="employee.JPG"
Content-Type	image/jpeg
Data	
BLOB	

14.	In variables, expand Environment. Note that the Environment folder now has a folder called Variables, with an element called Image. 
15.	Step over **twice** more. Two ESQl statements have been executed, to set the OutputRoot.JSON.Data.Employee and EmployeeSupplementary elements. ``` -- Set the Message Tree - Employee element set OutputRoot.JSON.Data.Employee = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.Employee; -- Set the Message Tree - EmployeeSupplementary element set OutputRoot.JSON.Data.EmployeeSupplementary = OutputRoot.MIME.Parts.Part[1].Data.JSON.Data.EmployeeSupple -- Make special arrangements for the binary image, which we want to store in the database as Base64 encoded. Set OutputRoot.JSON.Data.EmployeeSupplementary.IMAGE = BASE64ENCODE(Environment.Variables.Image); -- Save EMPNO in Env, so that the later map can send an appropriate message back to the client. set Environment.Variables.EMPNO = OutputRoot.JSON.Data.Employee.EMPNO; ```

16. The OutputRoot message now has a JSON folder as a primary folder in the JSON domain. It contains the full data of the EMPLOYEE and EMPLOYEE_SUPPLEMENTARY elements.

Note that the IMAGE element is not yet populated.

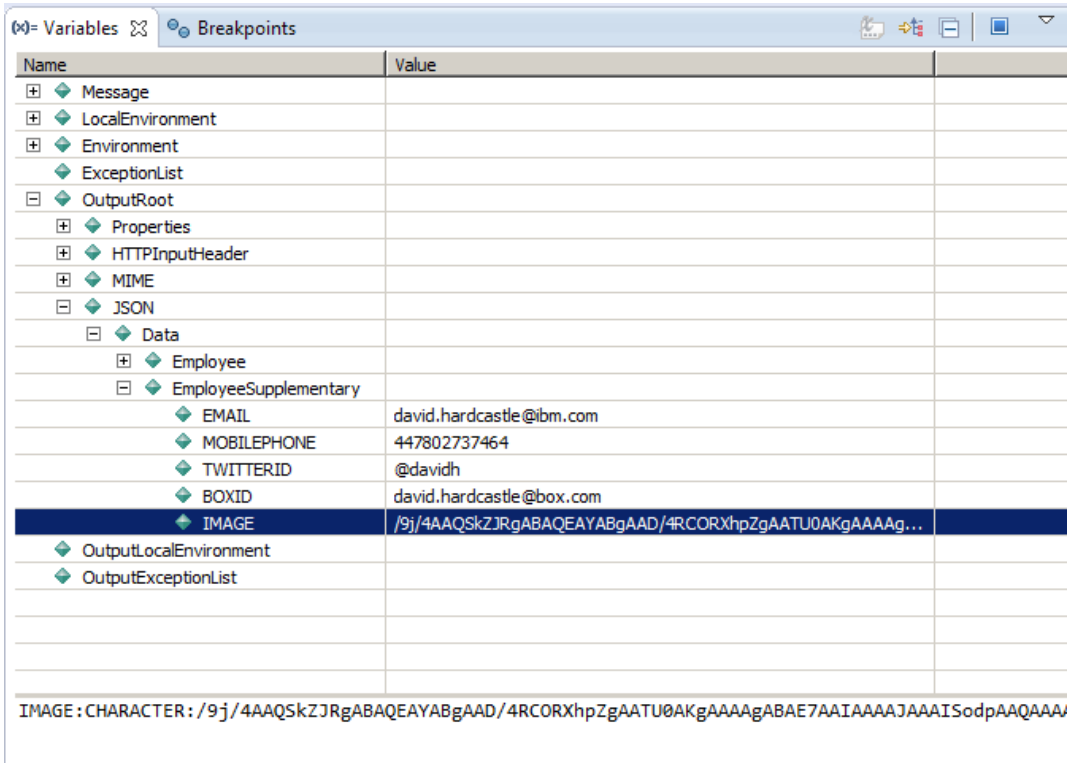


17. Step over once more.

```
-- Make special arrangements for the binary image, which we want to store in the database as Base64 encod
Set OutputRoot.JSON.Data.EmployeeSupplementary.IMAGE = BASE64ENCODE(Environment.Variables.Image);

-- Save EMPNO in Env, so that the later map can send an appropriate message back to the client.
set Environment.Variables.EMPNO = OutputRoot.JSON.Data.Employee.EMPNO;
```

18. Inspecting the Variables once more will show that the IMAGE element has been populated, and in fact has been converted to a Base64 encoded version of the binary image.

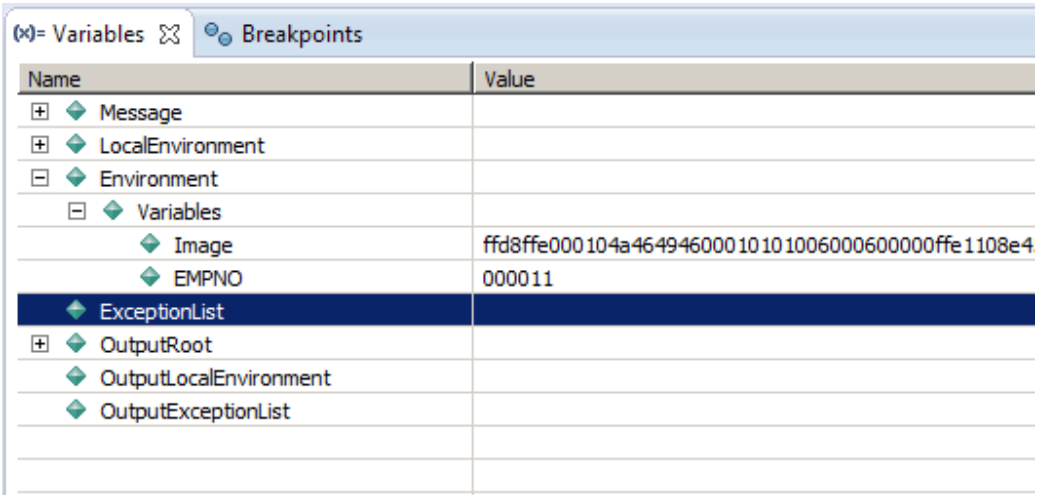


19. Step over once more.

```
-- Save EMPNO in Env, so that the later map can send an appropriate message back to the client.
set Environment.Variables.EMPNO = OutputRoot.JSON.Data.Employee.EMPNO;

set OutputRoot.MIME = null;
RETURN TRUE;
```

The Variables now show that the Environment contains a new variable EMPNO.

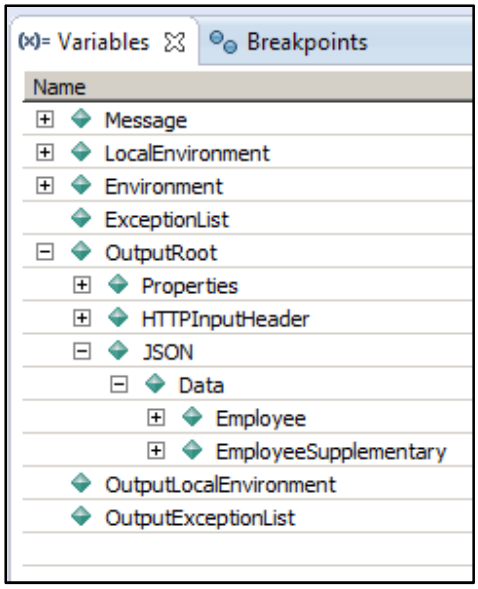


20. Step over once more.

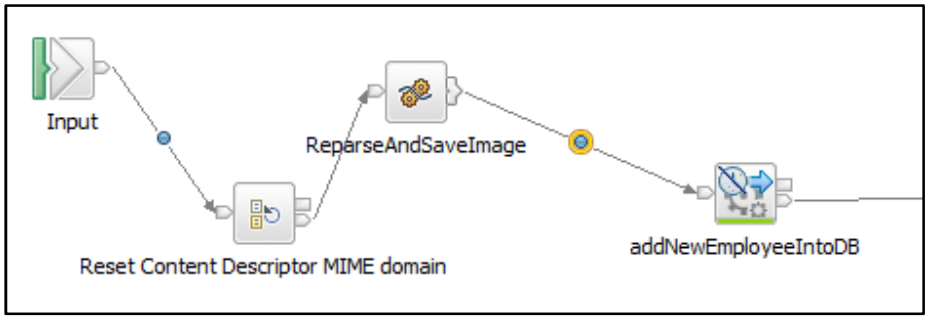
```
    -- Save EMPNO in Env, so that the later map can send an appropriate message
    set Environment.Variables.EMPNO = OutputRoot.JSON.Data.Employee.EMPNO;

    set OutputRoot.MIME = null;
    RETURN TRUE;
END;
```

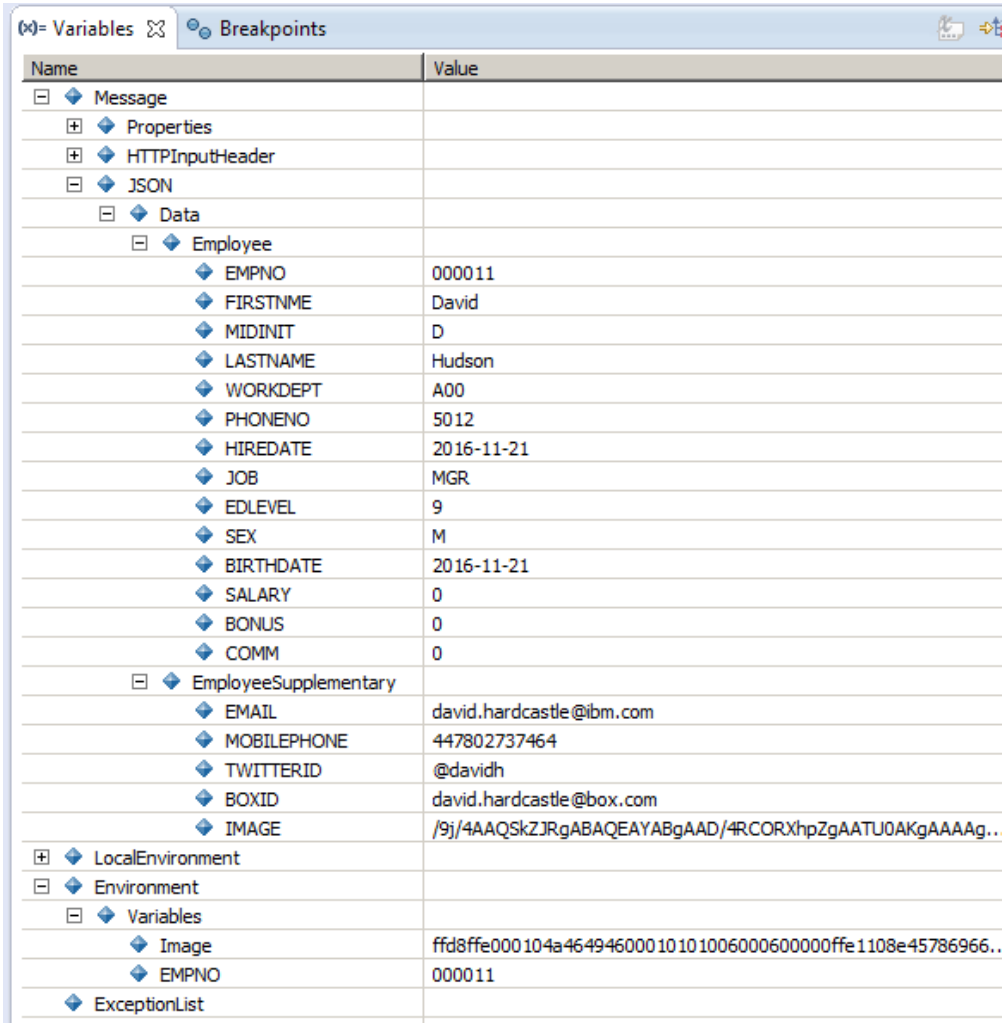
The message tree is now in the format that is required for normal JSON processing. The message body is in the JSON domain, and no other domains (eg. MIME) are present in the message.



21. Step over once more. The ESQL compute node will complete, and flow execution will resume. The flow will stop at the next node breakpoint.

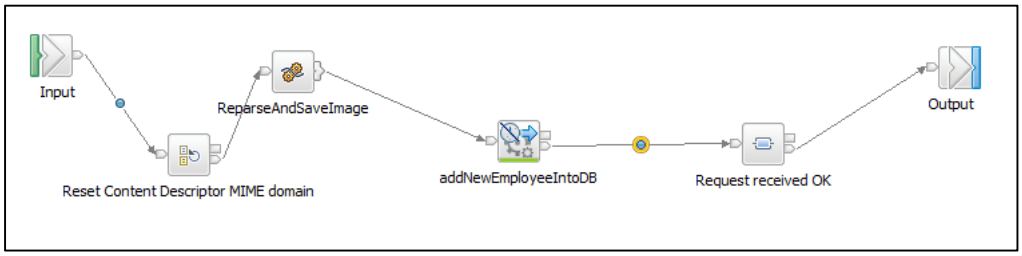
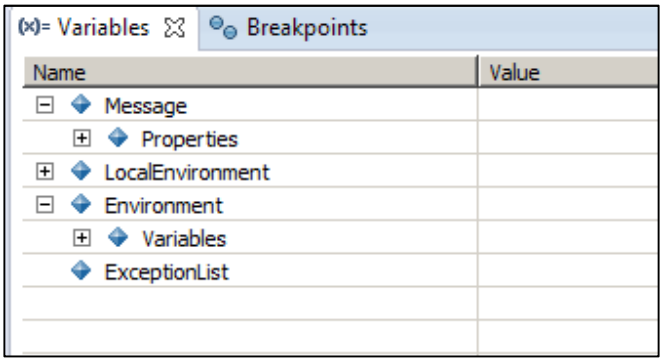


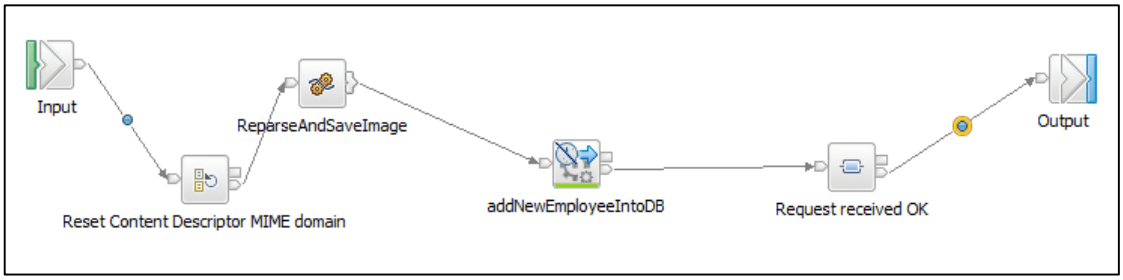
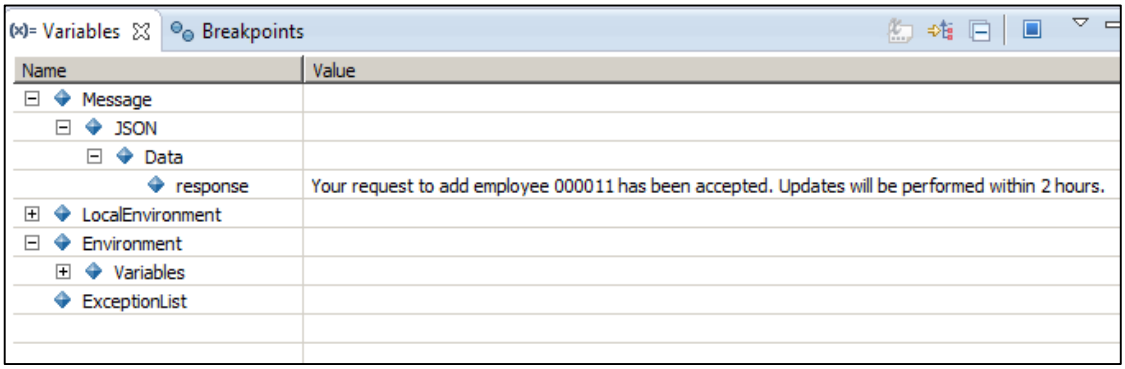
22.	In the Variables view, you will see that the flow has now extracted the JSON part of the message, and this is now held in the message tree, directly under the JSON folder. Additionally, the attached JPG image has been extracted, and is located in the Environment tree, under Variables/Image, in binary format. So, we now have the incoming message split into its two parts. The JSON part now represents the complete message. Additionally, the IMAGE has been converted from a binary attachment to a Base64 encoded element, contained in the main message body.

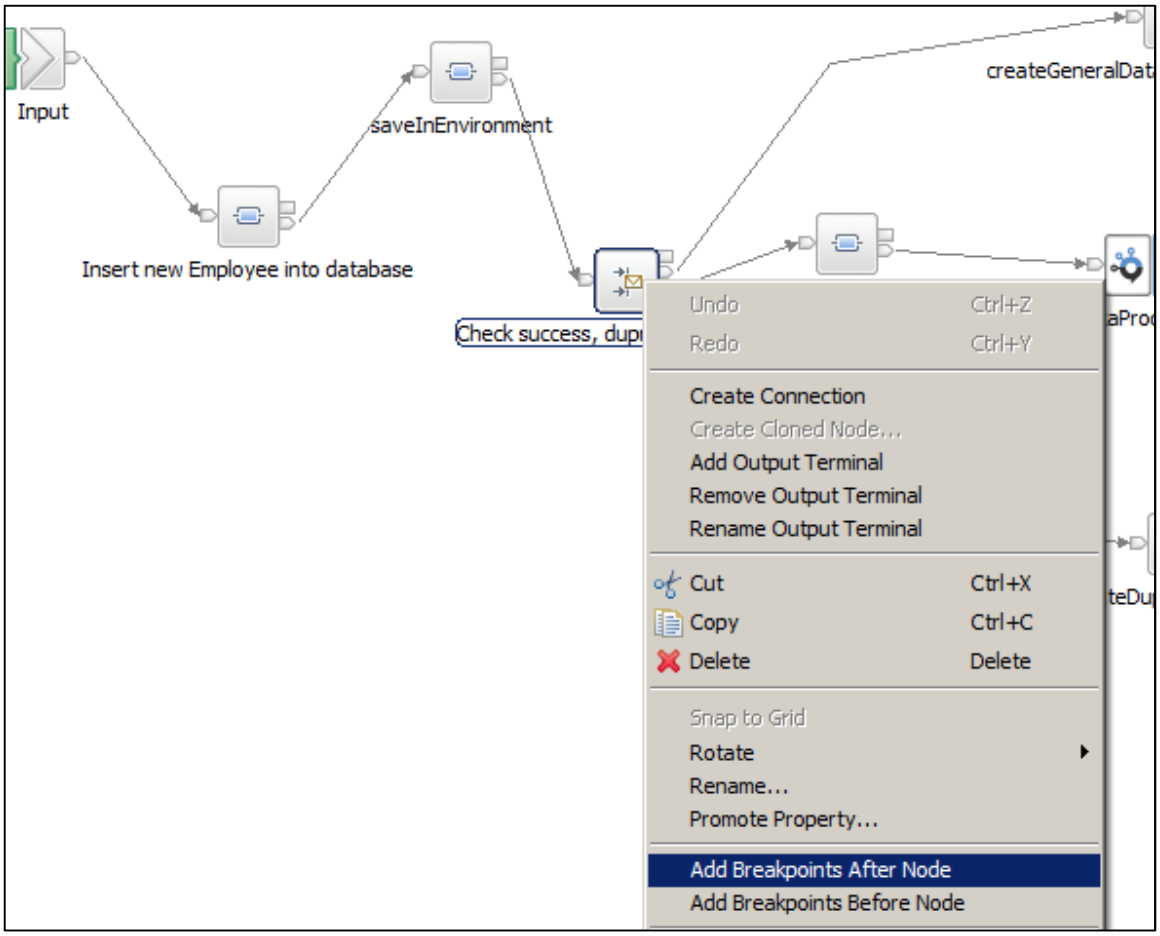


1.6.1 Execute the remainder of the flow

The remainder of this lab guide will not explicitly show the execution of every ESQL statement, although you are welcome to do so in your own testing.

1.	Step over the node after the addNewEmployeeIntoDB node. 
2.	The Variables now show that there is no message body (OutputRoot) at this point. 

3.	Step over once more.  The output message now contains a simple text message built by the map node. 
4.	Click Step Over to complete execution of the flow. If you have taken more than 180 seconds to execute this flow, the debugger will probably terminate. Click the debugger terminate buttons.

5.	The same process can be used to debug the addNewEmployeeIntoDB subflow. Note that when setting breakpoints, it is only necessary to set a breakpoint at the start of the flow. However, when a node that has multiple output terminals is used, explicit breakpoints should be set after each such node, as shown by right-clicking the “Check success, duprec or DB failure” Route node: 

2. Part 2 – Distributing Workload using Callable Flows

The Callable Flows feature enables the ability to split message flow processing between locations in a call/return (blocked wait) programming model.

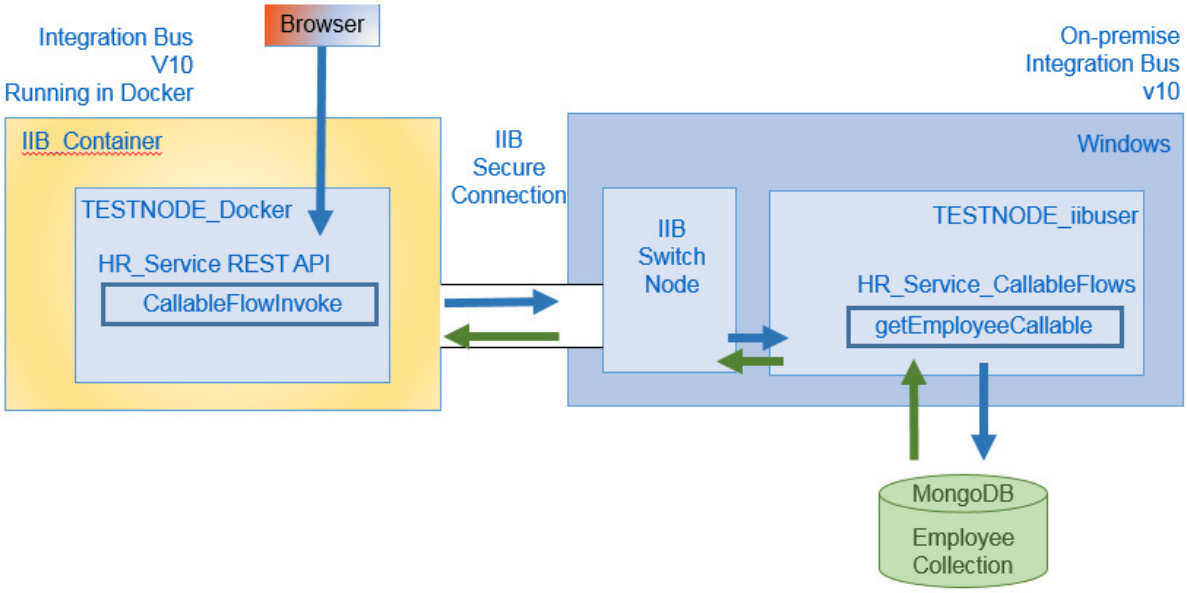
The CallableFlowInvoke node in a calling flow calls the CallableInput node of a callable flow. For example, a REST API running on IIB on Cloud can use a CallableFlowInvoke node to call a message flow (contained in an Application) running locally on premises. The message flow running locally uses a CallableInput node to receive input data and a CallableReply node to return data to the REST API on IIB on Cloud.

2.1 Scenario Overview

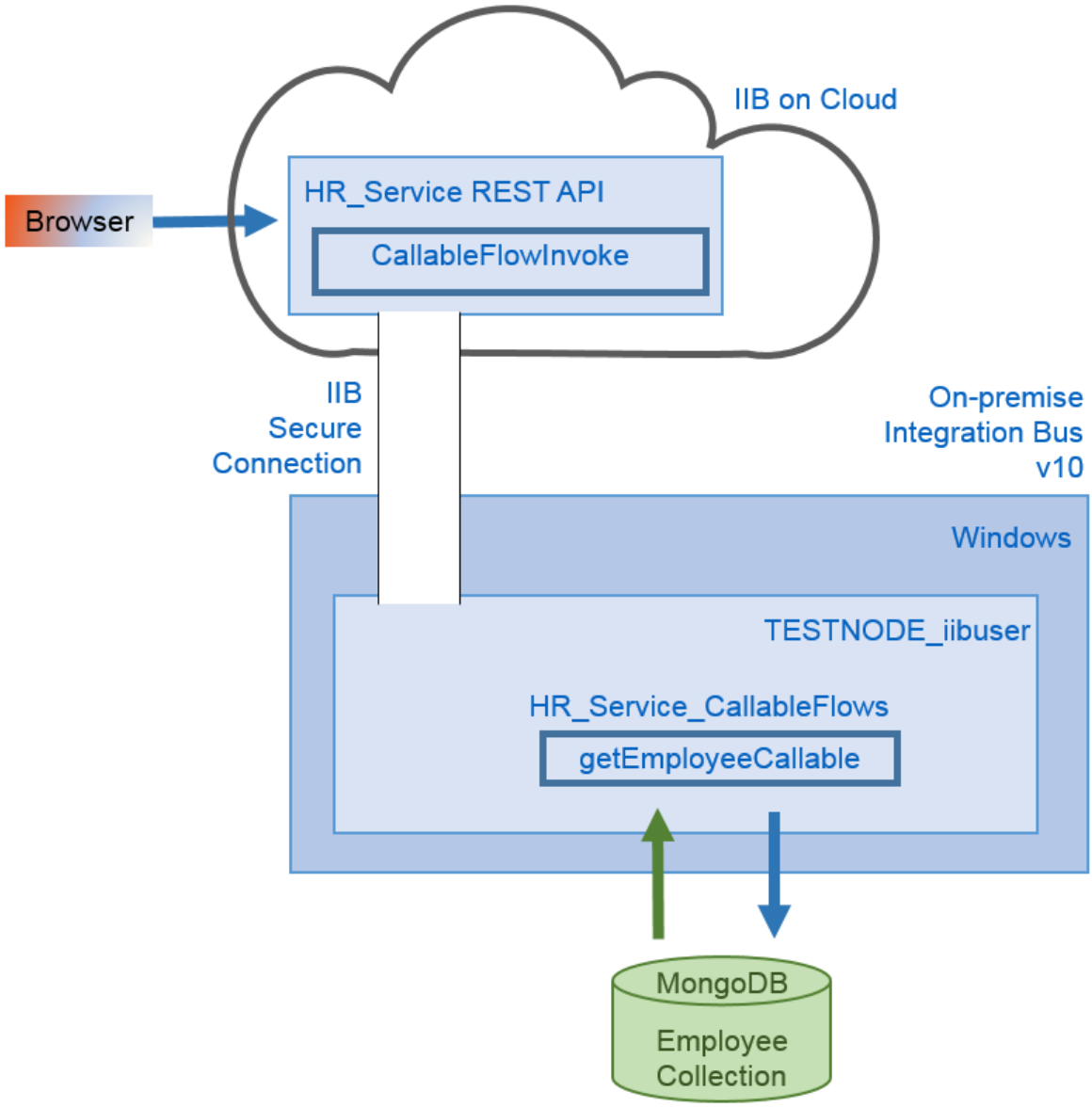
In this part of the lab you will explore the Callable Flows feature by configuring a REST API called HR_Service with an operation that uses a CallableFlowInvoke node to access a remote message flow.

The REST API (HR_Service) will be deployed in two remote locations:

a) Scenario 1: an IIB environment running in a Docker Container.



b) Scenario 2: the IBM service managed IIB on Cloud environment.

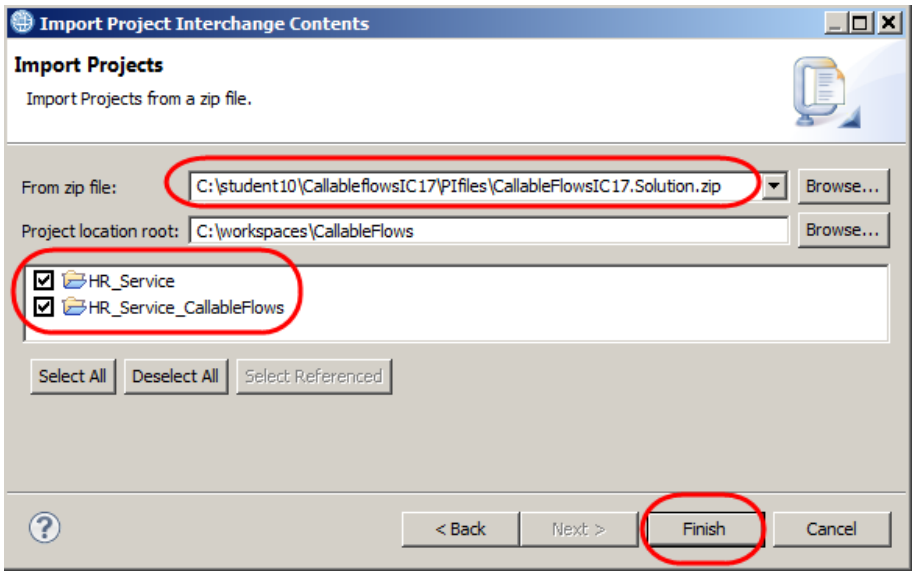
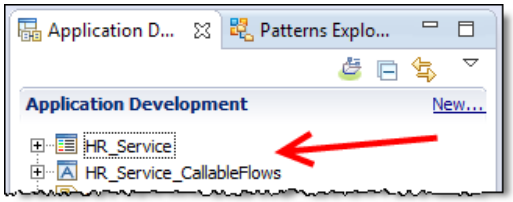


In each scenario the callable flow (known as HR_Service_CallableFlow) is running locally in your Windows environment and will return information from a NoSQL database using the IIB LoopBackRequest node.

2.2 Import Resources

The REST API and Callable Flow Application are provided for you. These are contained in a project interchange file which you will import into the Integration Toolkit.

The majority of tasks in this part of the lab are concerned with the configuration of the various components that will enable the REST API and Callable Flow to execute in different locations, and to access the MongoDB database.

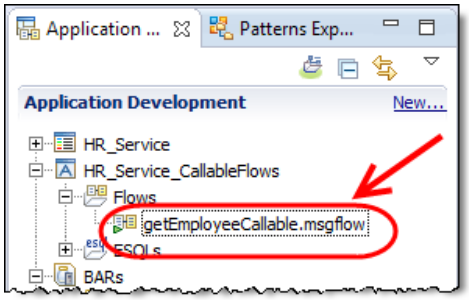
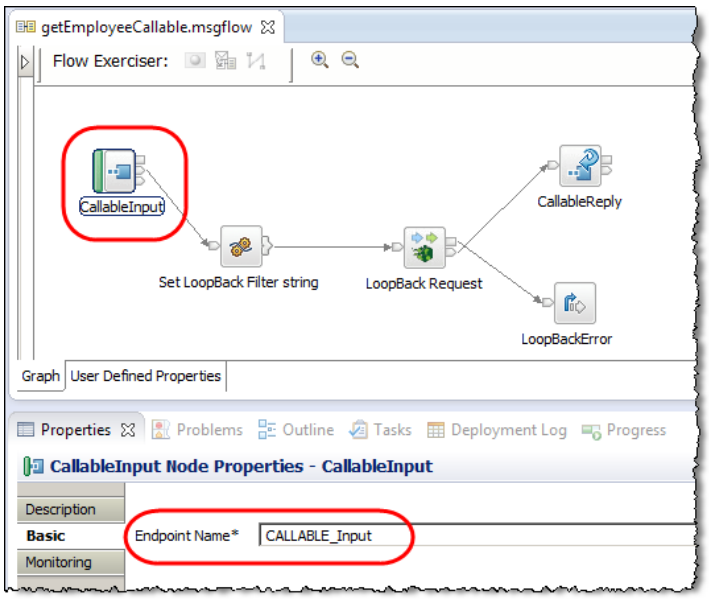
1.	The version of HR_Service that is used in this part of the lab is slightly different from that used earlier, so you will create a new workspace in which to import the solution files. In the Integration Toolkit, click File, Switch Workspace. Give the new workspace the name "CallableFlows", or similar.
2.	Right-click in the Application Development window and click 'Import'. Select Project Interchange, then click Next. Use the Browse button to navigate to `c:\student10\CallableFlowsIC17\PIfiles.` Select `CallableFlowsSolutionIC17.Solution.zip` Select All, then Finish: 
3.	HR_Service REST API and HR_Service_CallableFlows will now appear in your Application Development window: 

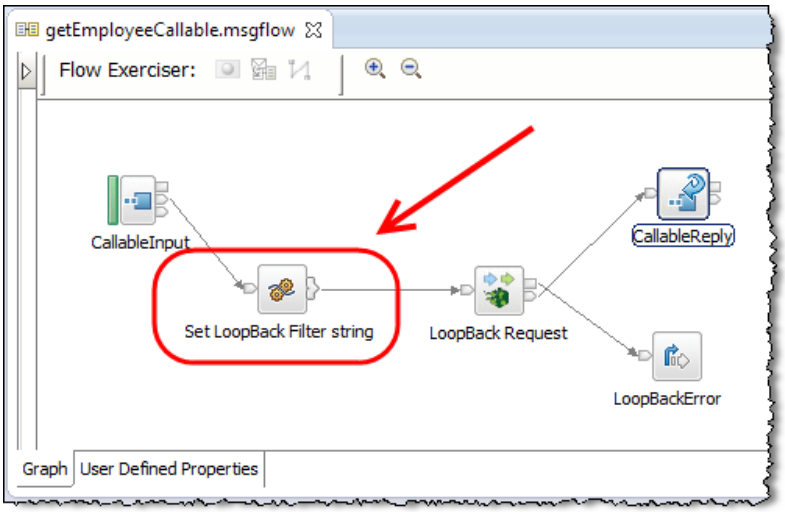
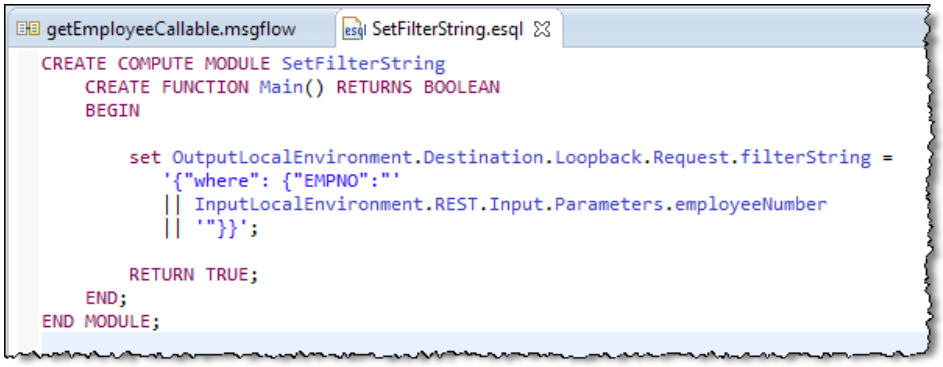
2.3 Review the Solution

In this next section you will review the REST API HR_Service and HR_Service_CallableFlows.

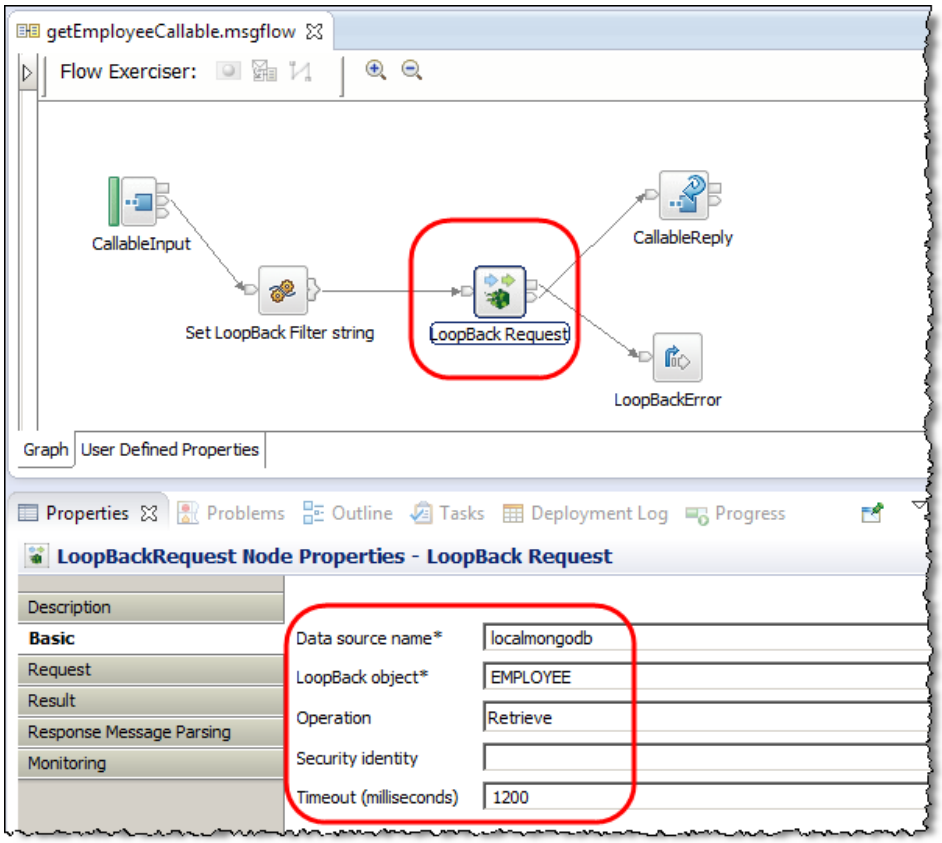
In this scenario the REST API HR_Service uses a CallableFlowInvoke node to call a callable flow running on the integration node in the Windows environment.

2.3.1 Review the getEmployeeCallable message flow

1.	In the Integration Toolkit, expand HR_Service_CallableFlows and then open getEmployeeCallable in the message flow editor: 
2.	Click the CallableInput node and review the node properties:  Note the Endpoint Name is set to `CALLABLE_Input`. This value is also specified on the Target Endpoint Name in the CallableFlowInvoke node in the calling message flow (see below).

3.	Open the ESQL node “Set LoopBack Filter string” : 
4.	The purpose of the ESQL node is to set the filter string used by the LoopBackRequest node to include the employeeNumber passed in the URL when the REST API HR_Service is called. This ensures that only documents relevant to the request are returned by the LoopBackRequest node:  Close the ESQL editor without saving any changes.

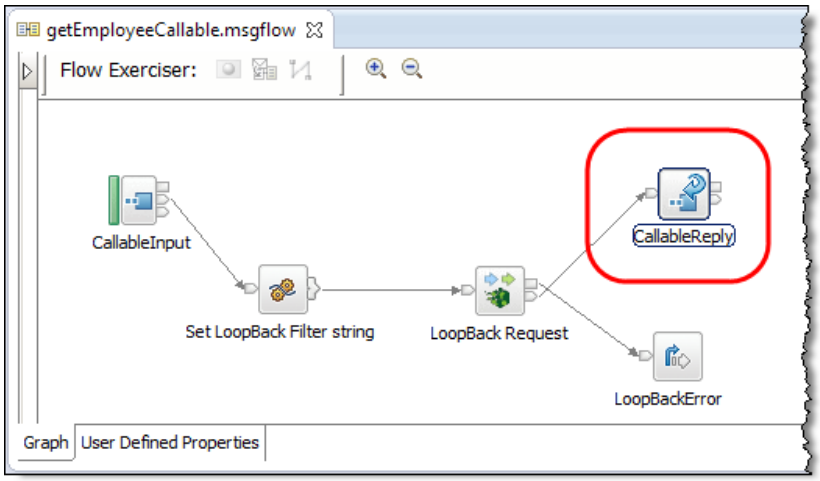
5.	Click the LoopBackRequest node and review the node properties tab:



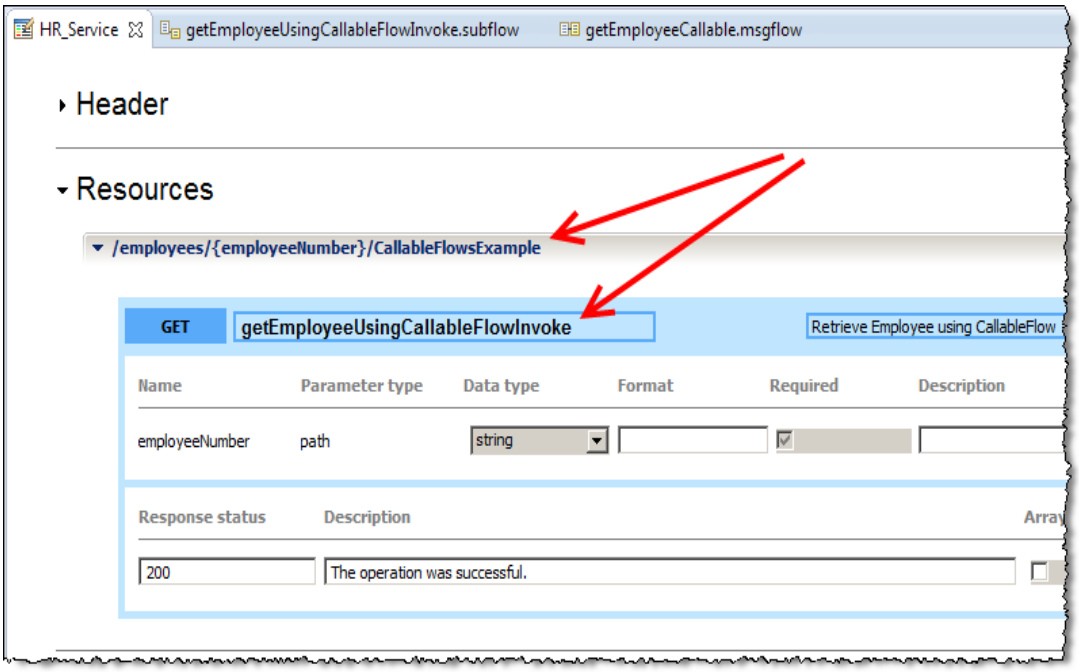
The data source name “localmongodb” is configured in datasources.json in C:\ProgramData\IBM\MQSI\connectors\loopback\. The file enables the message flow to access the local mongoDB environment (no further configuration of this file is required).

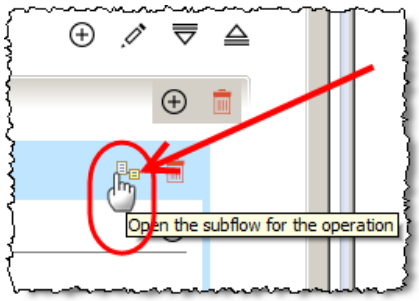
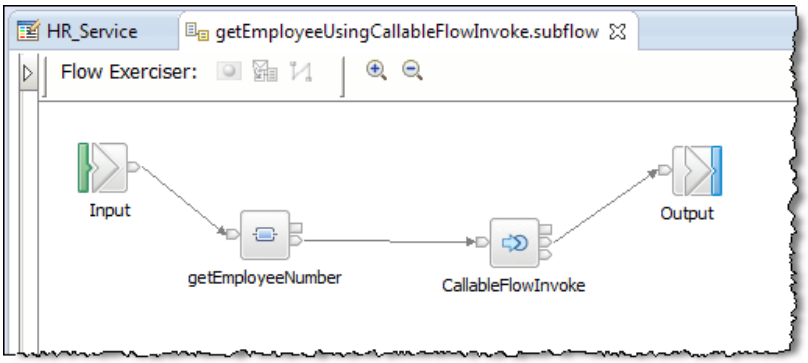
The LoopBack Object field is set to EMPLOYEE. The HRDB database in mongodb has been pre-loaded with employee documents in json message format.

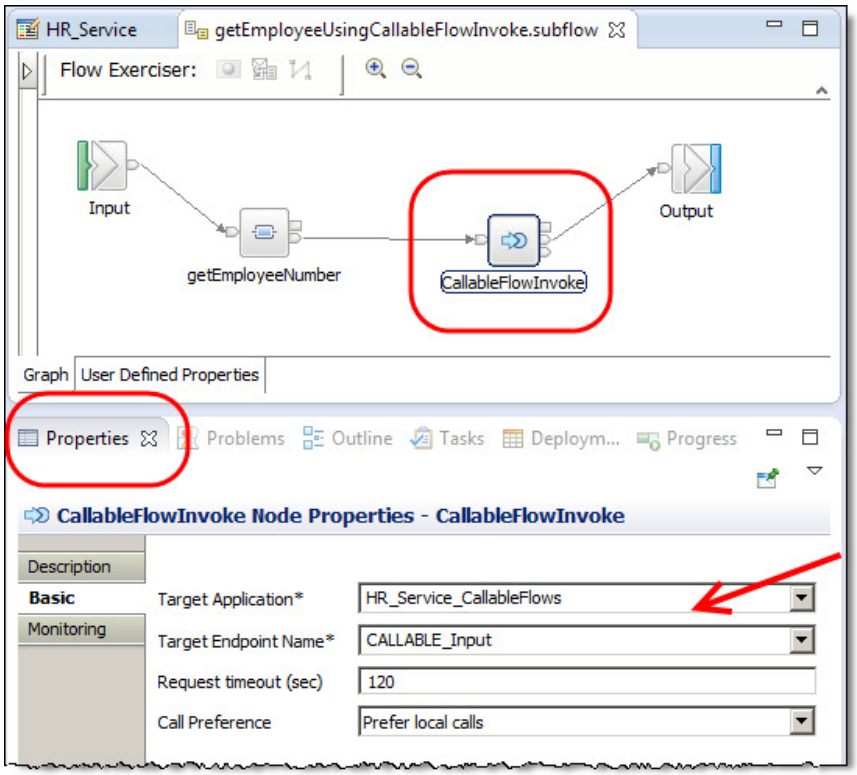
Note the Loopback operation is set to Retrieve.

6.	The CallableReply node passes control back to the CallableFlowInvoke node 

2.3.2 Review the getEmployeeUsingCallableFlowInvoke operation

1.	Open the HR_Service REST API Description. Expand Resources and note the getEmployeeUsingCallableFlowInvoke operation in /employees/{employeeNumber}/CallableFlowsExample resource: 

2.	Scroll to the right and click the open the subflow for the operation: 
3.	The getEmployeeUsingCallableFlowInvoke subflow will open: 
4.	The getEmployeeNumber mapping node stores the employeeNumber passed in the REST API URL in the message tree, ready to be passed to the CallableFlowInvoke node.

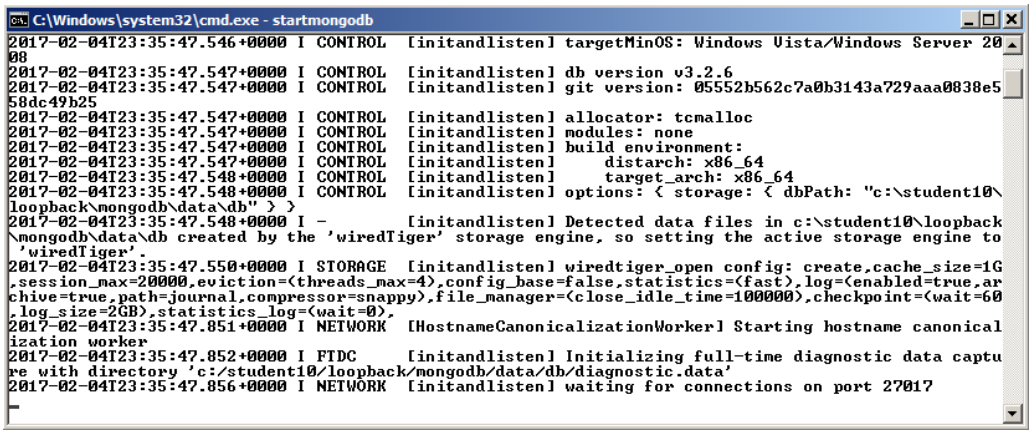
5.	Click the CallableFlowInvoke node and select the node properties tab:  Note: the Target Application is set to HR_Service_CallableFlows (*the application you imported*). The Target Endpoint Name is set to "`CALLABLE_Input`".
6.	Close the HR_Service REST API and the getEmployeeCallable message flow.

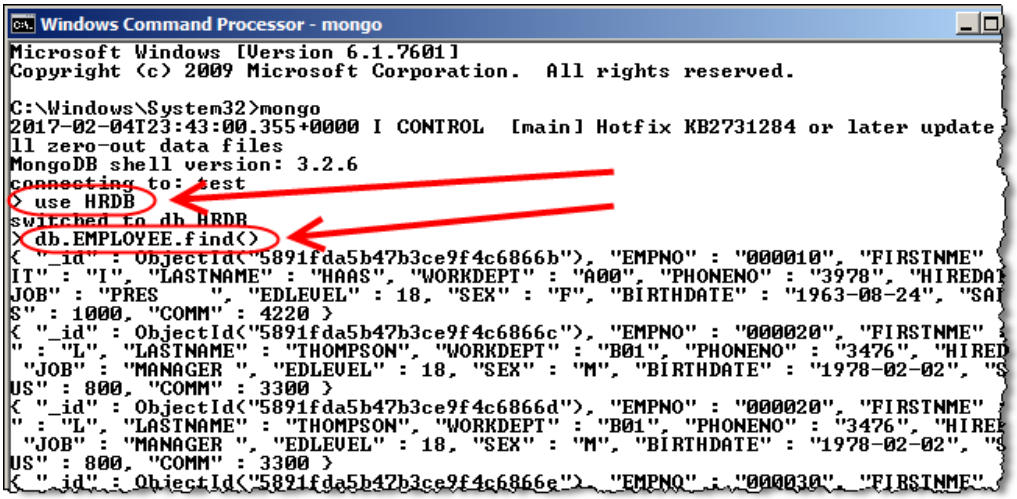
2.4 The NoSQL database

The function of getEmployeeCallable is to provide information from a local noSQL Database. In this scenario the database is MongoDB. In this next section you will prepare the MongoDB environment ready to be used in this scenario. The MongoDB environment has already been pre-configured for you in this lab environment. The MongoDB loopback connector has been installed into the IIB environment you are using. For more information on configuring IIB to use the LoopBack node refer to the online Knowledge Center.

2.4.1 Start MongoDB

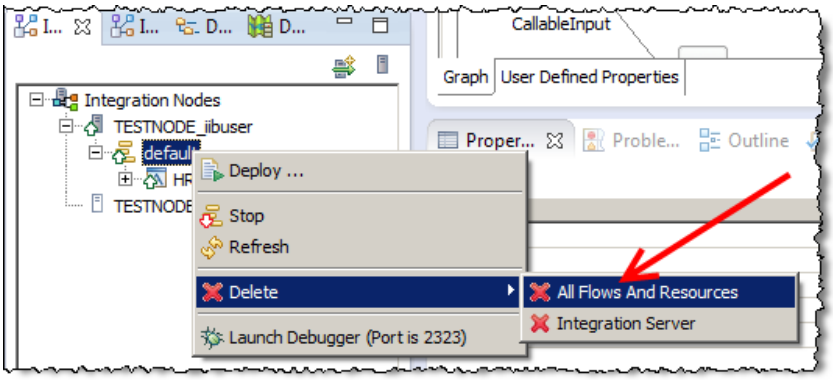
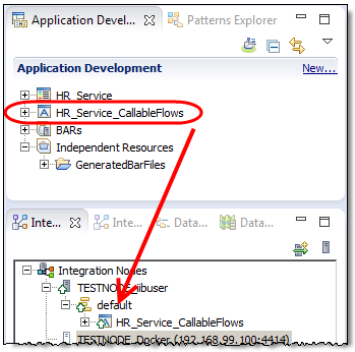
1.	If there are Windows command prompts open from the previous part of this lab please close them all now.
2.	In a Windows Command Prompt, navigate to : `c:\student10\Loopback\mongodb\commands` Run the command: `startMongoDB` For info, this will run the MongoDB command: **`mongod.exe --dbpath c:\student10\Loopback\mongodb\data\db`** This command will start the MongoDB server. No defaults have been changed, so the MongoDB server will start with the client listener on port 27017. The command window will be held open. ***Do not close this window, if the window is closed the MongoDB server will terminate.***



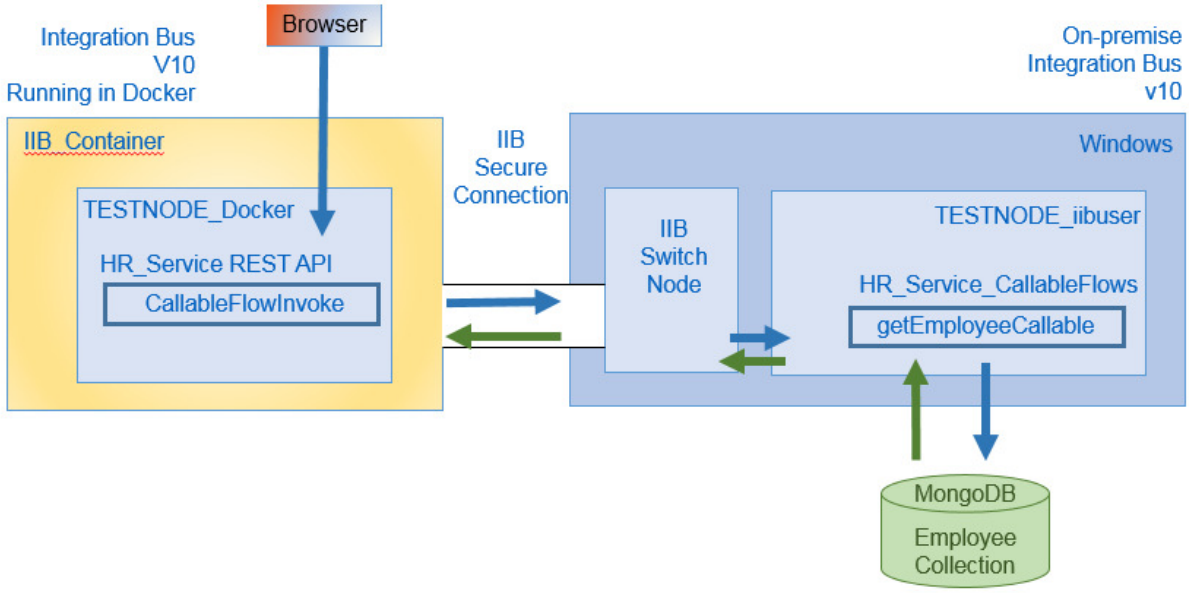
3.	Start a Mongo client shell. Open a new Windows Command Prompt, and execute the command "**mongo**". This will use the default port of 27017, and connect to the started server. Note: that the mongo client will initially connect to the server, and will connect to the **test** database.
4.	To verify the MongoDB server is working correctly enter the following commands. To switch to the HRDB database: `use HRDB` To list all documents in the EMPLOYEE collection: `db.EMPLOYEE.find()` 

2.5 Deploy HR_Service_CallableFlows

In this scenario the REST API HR_Service uses a CallableFlowInvoke node to call a callable flow running on the integration node in the Windows environment.

1.	In the Integration Nodes view, right click on the default integration server and select Delete > All Flows and Resources to avoid conflicts with earlier scenarios. 
2.	In the Integration Toolkit drag and drop the HR_Service_CallableFlows application onto the default server in TESTNODE_iibuser: 

2.6 Scenario:1 Running CallableFlowInvoke in an IIB Docker Container



A Docker container with IIB V10.0.0.7 is provided for you on the Windows system you are using. The Docker container runs in an Ubuntu Linux environment (hosted in the Windows system using Oracle VirtualBox). In this next section you will start the Ubuntu Linux image and the IIB Docker Container in preparation for deploying the calling REST API.

The Docker container is configured with an IIB integration node TESTNODE_Docker. For information purposes (***do not execute these commands***) the container was created using the following method:

- Obtain a docker file and scripts from https://github.com/ot4i/iib-docker
- Build a Docker image called **iibv10007image** using:

```
docker build -t iibv10007image .
```

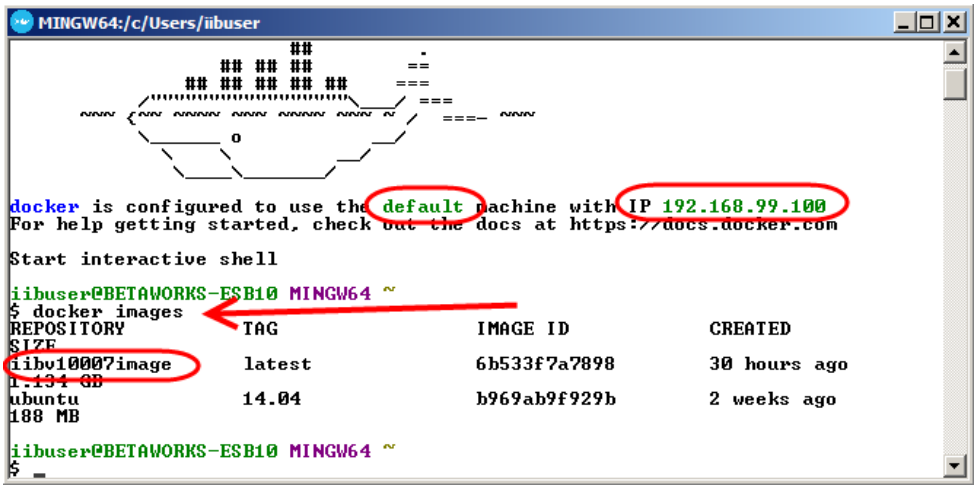
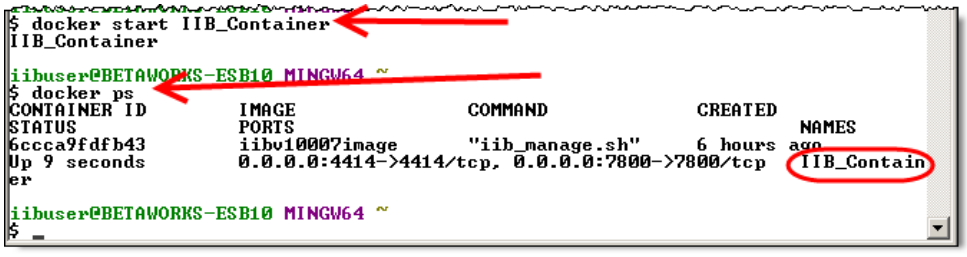
- The **iibv10007image** is then used to create a Docker container called **IIB_Container** an integration node name of TESTNODE_Docker and default IIB port values exposed using:

```
docker run --name IIB_Container
     -e LICENSE=accept
     -e NODENAME=TESTNODE_Docker
     -p 7800:7800
     -p 4414:4414
     -h BETAWORKS-ESB10-DOCKER
     iibv10007image
```

The Docker build and run comands can take some time to complete. In order to use the IIB environment running in Docker, the Docker start command is used to start IIB_Container.

2.6.1 Start the IIB Docker container

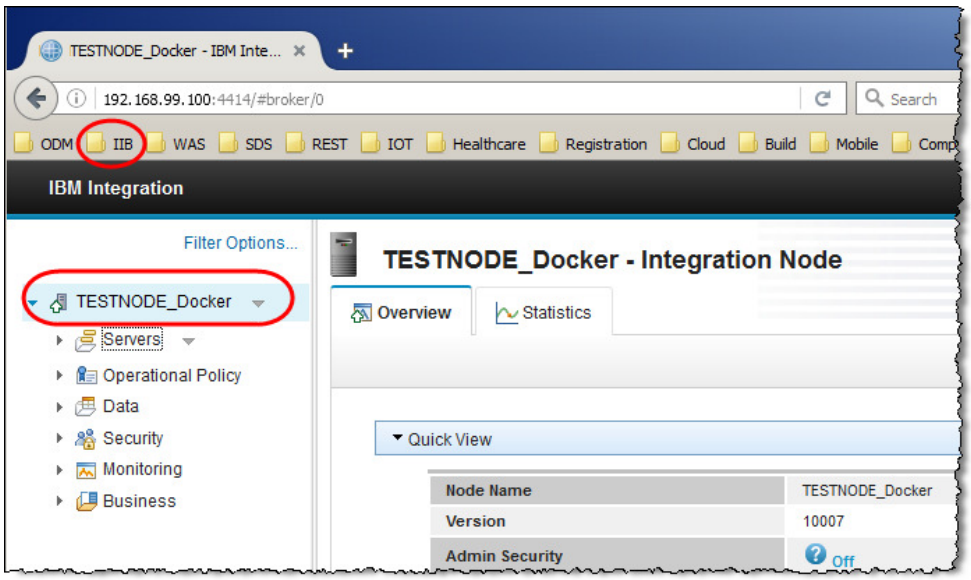
1.	Open a Docker command prompt by double-clicking on the Docker Quickstart Terminal icon  When you open the Docker Quickstart Terminal a terminal window will open. If the default Ubuntu Linux VM image managed and controlled by Oracle VM VirtualBox is not started, it will be started automatically. The Docker technology is not the focus in this lab, for more information on Docker is available at ***https://www.docker.com.***

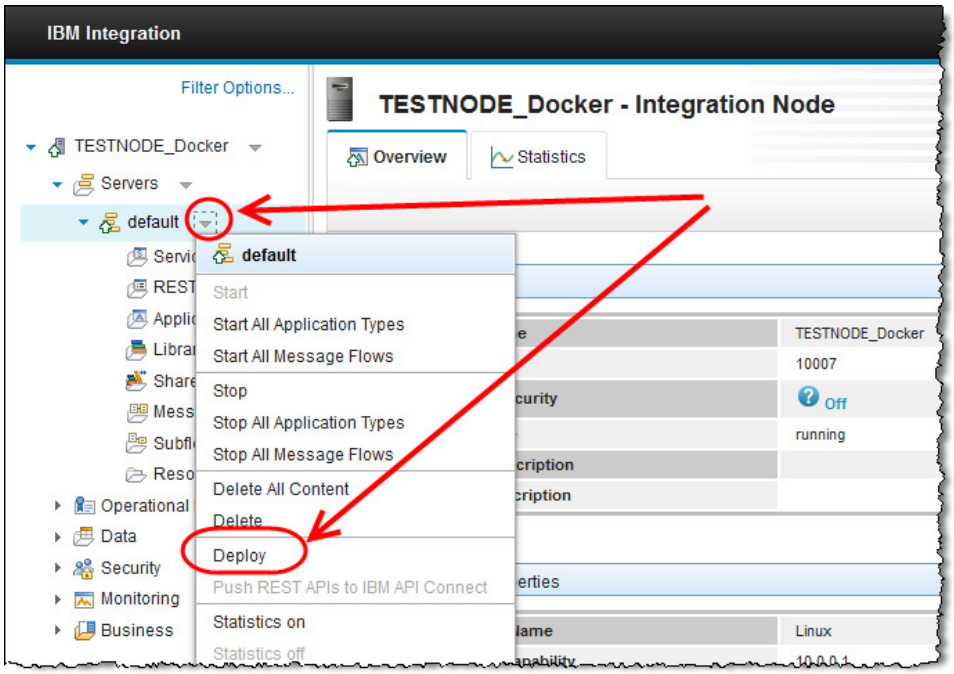
2.	When the terminal opens, you will see the details of the default Ubuntu VM. In this example, the name of the VM is "`default`" and the IP address is `192.168.99.100`. To see the list of docker images type: `docker images` 
3.	List the current Docker container using: `docker ps -a` The `-a` will show the container if it is not started.
4.	Start the docker container with IIB installed by entering the command: `docker start IIB_Container`
5.	Verify the docker container is running by entering the command: `docker ps` 

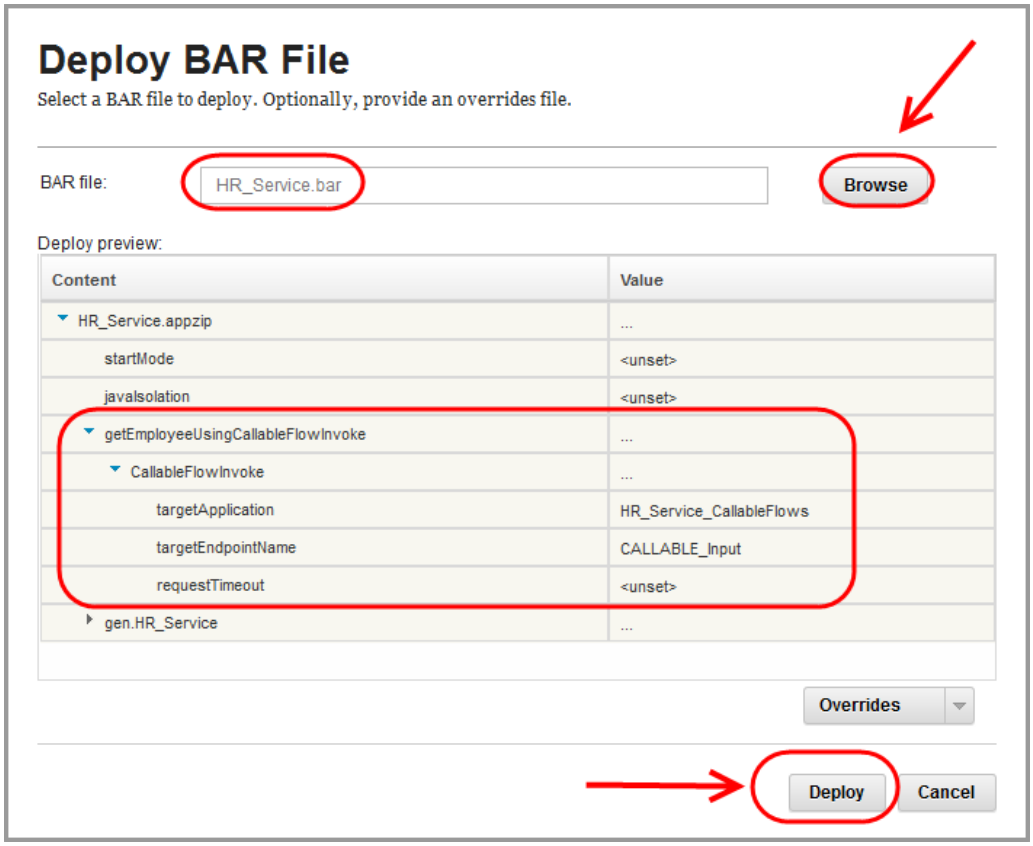
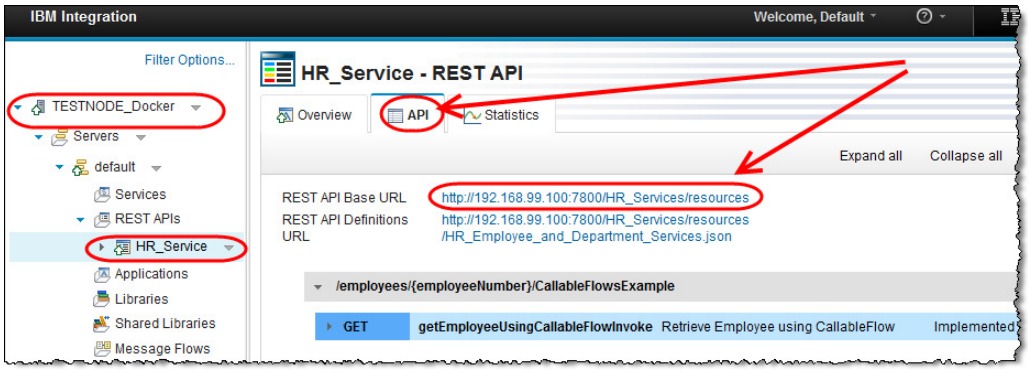
2.6.2 Deploy HR_Service in TESTNODE_Docker

TESTNODE_Docker will now be running in the IIB_Container in the hosted Linux environment. In this next section you will use the IIB Web Admin interface to deploy the HR_Service REST API on TESTNODE_Docker.

1.	Open a Firefox browser and select the “IIB Docker” link (in the IIB folder). This will direct the browser to the Web Admin interface for the IIB node `TESTNODE_Docker` that is running in the Docker container:



2.	On the left navigation bar, expand Servers. Click the triangle next to the “default” server and select Deploy: 
3.	In the Deploy Bar file window, use the Browse button to navigate to `c:\student10\CallableflowsIC17\barfiles` and select `HR_Service.bar`.

4.	In the Deploy preview window, note the details of the `targetApplication` and `targetEndpointName` in the CallableFlowInvoke node. Click Deploy: 
5.	After a few seconds a green message will appear detailing that the deploy was successful. Reload the page using the browser refresh button.
6.	Select the HR_Service REST API in the left navigation window. In the right window, click the API tab and copy the REST API Base URL: 

7.	Open a Firefox tab and paste the REST API Base URL into the browser. Append "`/employees/000010/CallableFlowsExample`" onto the URL and press enter.  The CallableFlowInvoke node running on TESTNODE_Docker cannot locate the getEmployeeCallable message flow running in the Windows environment (BETAWORKS-ESB10). In order for the CallableFlowInvoke node to establish a connection with the getEmployeeCallable message flow (in HR_Service_CallableFlows), a secure trusted connection must be configured on both sides. In the next section you will configure a callable flow agent to enable the communication between the two flows. **Leave this Browser window open. You will re-test the URL when the IIB Switch configuration is complete.**

2.6.3 Create and configure IIB Switch on the Windows environment

In this part of the lab you will create an IIB Switch on the Windows IIB environment. This will enable a secure connection between the integration node running in your hosted Docker environment (TESTNODE_Docker) and the integration node running in Windows (TESTNODE_iibuser).

The Switch can be created on either of the IIB installations. In this scenario you will create the IIB Switch on the Windows environment.

The files generated will be stored in the folder **c:\temp**.

1.	Open an Integration Console and run the command: `iibcreateswitchcfg /hostname BETAWORKS-ESB10 /output c:\temp` The command will respond with the following messages: `Generated self signed certificate file 'c:\temp\adminClient.p12'` `Generated switch configuration file 'c:\temp\switch.json'` `Generated agentx configuration file 'c:\temp\agentx.json'`
2.	The command creates two JSON configuration files, and a certificate. • `adminClient.p12`: is a certificate used to store a private keys and certificate chain for the connection between the IIB Docker container and the 'On-premises' Integration Node. • `switch.json`: is used to create the Switch server. • `agentx.json`: is used by the **mqsichangeproperties** command to configure secure connectivity for the integration servers where your flows are deployed. The file is used to configure both IIB environments – IIB running on Windows and IIB running in the Docker container. The flag **`/hostname`** in the command above ensures that the above configuration files contain the hostname where the Switch has been created.
3.	Run the **iibswitch** command to create the Switch server by using the configuration file (`switch.json`) that you created in the previous step. **`iibswitch create switch /config c:\temp\switch.json`** You will see the following response: `Creating iibswitch component 'switch', please wait...` `iibswitch created and started.` If you receive the response "`iibswitch already created, cannot create`", rerun the command and replace `create` with `update`.
4.	To test that the Switch server is created and running, run the command **`mqsilist IIBSWITCH_NODE`** The response will read: `----------` `BIP1286I: Integration server 'IIBSWITCH_SERVER' on integration node 'IIBSWITCH_NODE' is running` `BIP8071I: Successful command completion.`

5.	You will need to ensure that the integration server where you have deployed your Callable Application has the correct certificate to communicate securely with the Switch server. This requires you to run the **mqsichangeproperties** command for each integration server where you have deployed callable message flows. The command uses the integration server configuration file (`agentx.json`) that you created in step 1. In an Integration Console, navigate to `c:\student10\CallableflowsIC17\commands\` and run the file: `ConfigureTESTNODE_iibuserDockerAgentX.cmd` Ensure the command, completes with `BIP8071I: Successful command completion.` For information the comand that this file runs is: `mqsichangeproperties TESTNODE_iibuser` `-e default` `-o ComIbmIIBSwitchManager` `-n agentXConfigFile` `-p c:\temp\agentx.json`
6.	Stop and restart the TESTNODE_iibuser on the Windows environment to make the changes effective: **`mqsistop TESTNODE_iibuser`** **`mqsistart TESTNODE_iibuser`**

2.6.4 Configure TESTNODE_Docker to use IIB Switch

In this part of the lab, you will configure TESTNODE_Docker running in the Docker container to access TESTNODE_iibuser in the Windows environment through a secure connection.

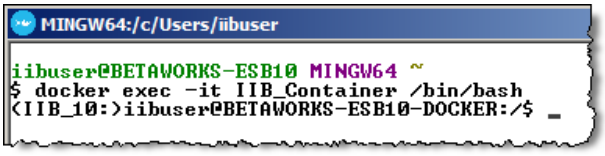
Now that you have the IIB Switch running on the Windows environment, the same `agentx.json` configuration file is required on TESTNODE_Docker. When this configuration is complete, both IIB nodes will be able access callable flows running on each other's environment, through a secure connection.

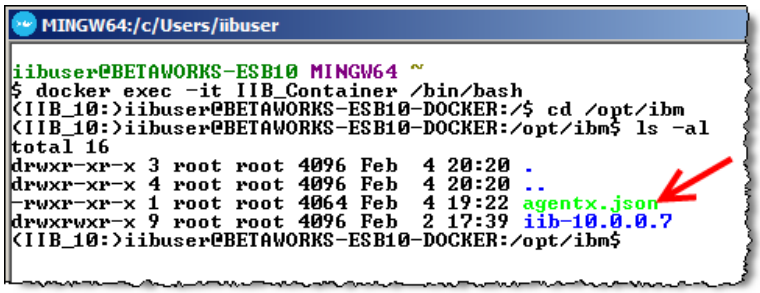
2.6.4.1 Copy agentx.json to the IIB Docker Linux File System

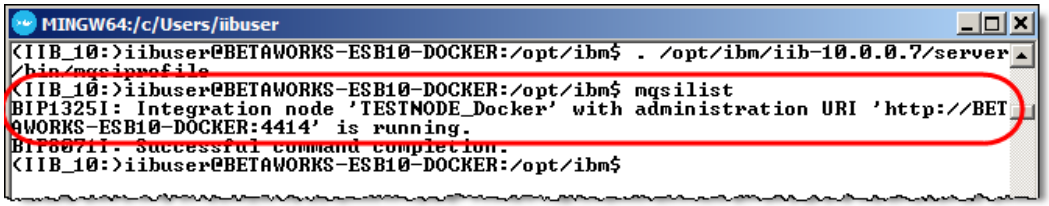
1.	Docker provides a command (**docker cp**) which will copy a file from a host to the Docker image (or vice versa). The configuration file `agentx.json` is in the `c:\temp` folder, which is a local folder on the VM. You will copy that file to the directory `/opt/ibm` in the Linux file system used by the Docker container **IIB_Container.** In the Docker Quickstart terminal, run the command (Note the forward slashes in all parts of this command): `docker cp c:/temp/agentx.json IIB_Container:/opt/ibm`  You will verify the copy in the following steps.

2.6.4.2 Configure Docker Quickstart Terminal for MQSI commands

In this section you will set up the Docker Quickstart terminal to run commands directly in the IIB_Container running in the hosted Ubuntu Linux environment.

1.	In the Docker Quickstart terminal, attach a bash session to the IIB_Container by running the command: **`docker exec -it IIB_Container /bin/bash`**  Commands you type in this terminal will now be executed in IIB_Container.

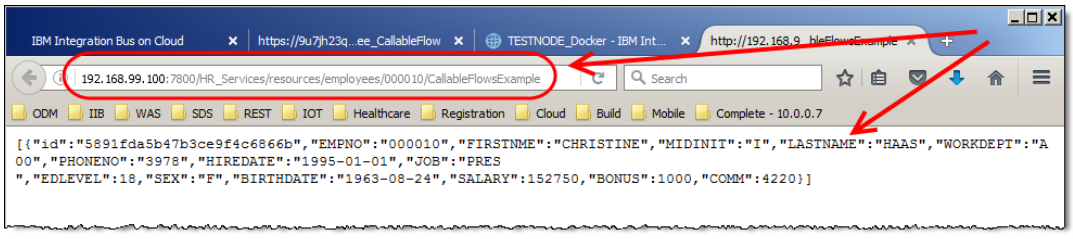
2.	Verify that the earlier copy of the `agentx.json` file was successful and is available in the container. If you followed the instructions the file should be in the `/opt/ibm` directory. Go to that directory to view its content. In the command prompt, run the command below. `cd /opt/ibm`
3.	List the /opt/ibm contents, by running the command: **`ls -al`**  You will see that the agentx.json file has been copied successfully and you should see the current date.
4.	Verify that IIB_Container can communicate with the Windows system using the hostname BETAWORKS-ESB10. Type: `Ping BETAWORKS-ESB10` If you receive a successful response from the ping command, proceed with the next steps. ********** If the hostname cannot be resolved, Refer to the Appendix at the back of this guide to add a host entry for BETAWORKS-ESB10 to /etc/hosts, then come back to the next instruction in this lab. **********

5.	When you start an IIB runtime component on Linux and UNIX systems, the runtime component will inherit the environment from where you issue the **mqsistart** command. You must therefore initialize the environment before you start a component; the command **mqsiprofile** performs this initialization The `mqsiprofile` command is located in the IIB directory `/opt/ibm/iib-10.0.0.8/server/bin`. In the Docker command prompt, run the command below (***note the dot at the beginning***). **`. /opt/ibm/iib-10.0.0.8/server/bin/mqsiprofile`** If the command is successful you will see no response.
6.	The Docker Quickstart terminal will now be capable of running **mqsi** commands. Run the command : **`mqsilist`** You will see that the command return confirmation that TESTNODE_Docker is up and running. 

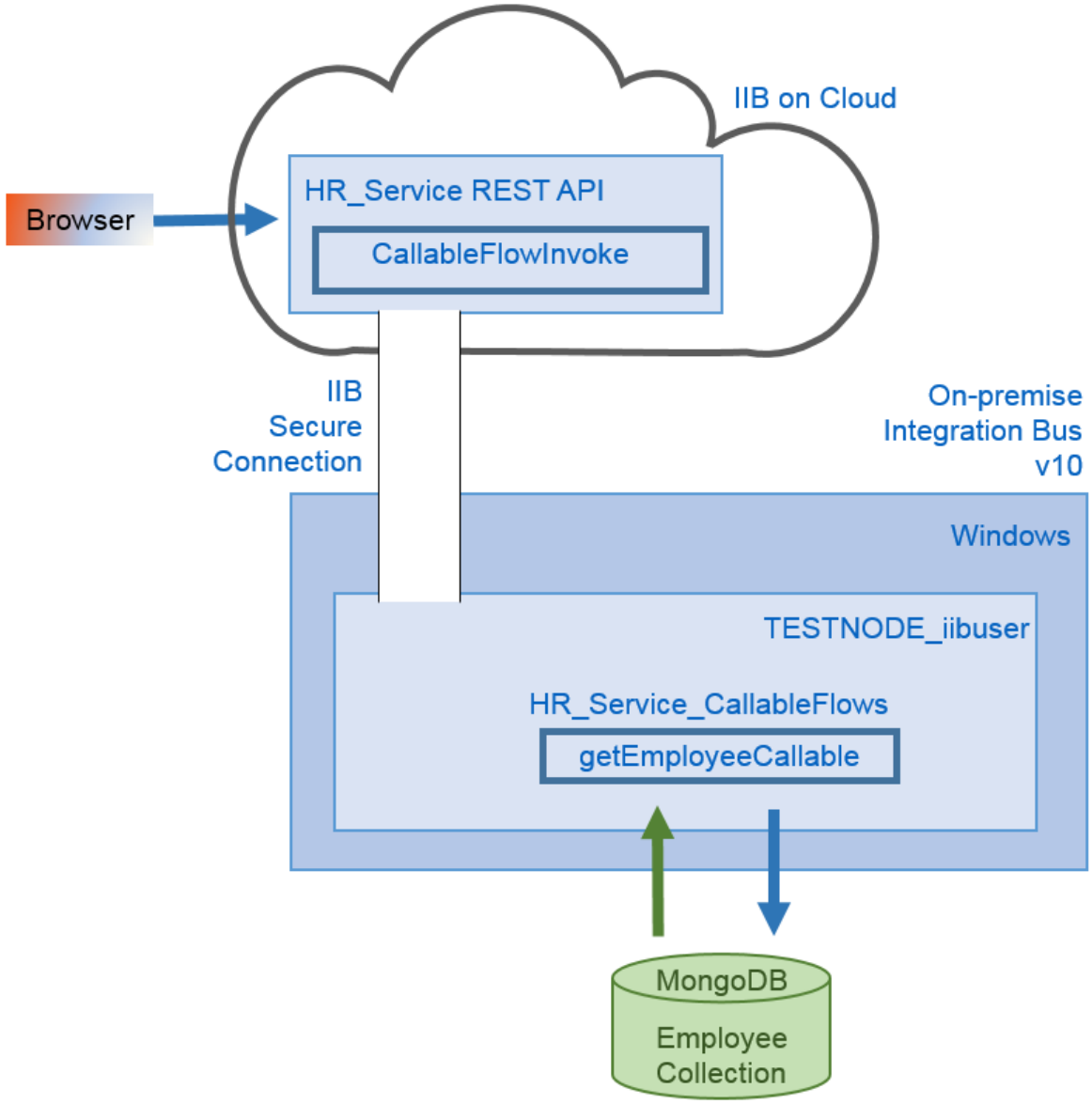
2.6.4.3 Configure TESTNODE_Docker to connect to IIB Switch

1.	In order to configure TESTNODE_Docker to connect to the IIB Switch on the IIB in your Windows environment, run the **mqsichangeproperties** command using the generated **agentx.json** configuration file. Run the command (*all on one line*) **mqsichangeproperties TESTNODE_Docker** **-e default** **-o ComIbmIIBSwitchManager** **-n agentXConfigFile** **-p /opt/ibm/agentx.json** The command will complete successfully with the response: `BIP8071I: Successful command completion.`
2.	Stop and restart the TESTNODE_Docker: **mqsistop TESTNODE_Docker** and **mqsistart TESTNODE_Docker**
3.	If you need access to the IIB syslog messages for TESTNODE_Docker. Run the command: `tail -f /var/log/syslog`

2.6.5 Re-test HR_Service running in TESTNODE_Docker

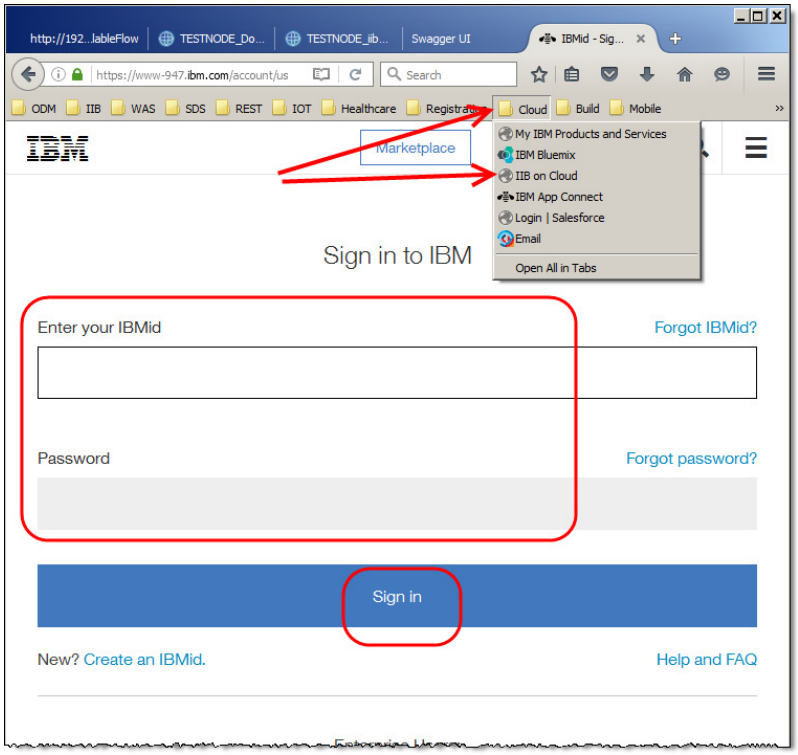
1.	In the Firefox browser you left open, press refresh on the URL (the URL you copied from the IIB Web Administration tool). The HR_Service REST API is now able to call getEmployeeCallable. The Callable Flow then uses the LoopBackRequest node to obtain data from the NoSQL database for the employee with EMPNO=000010. 

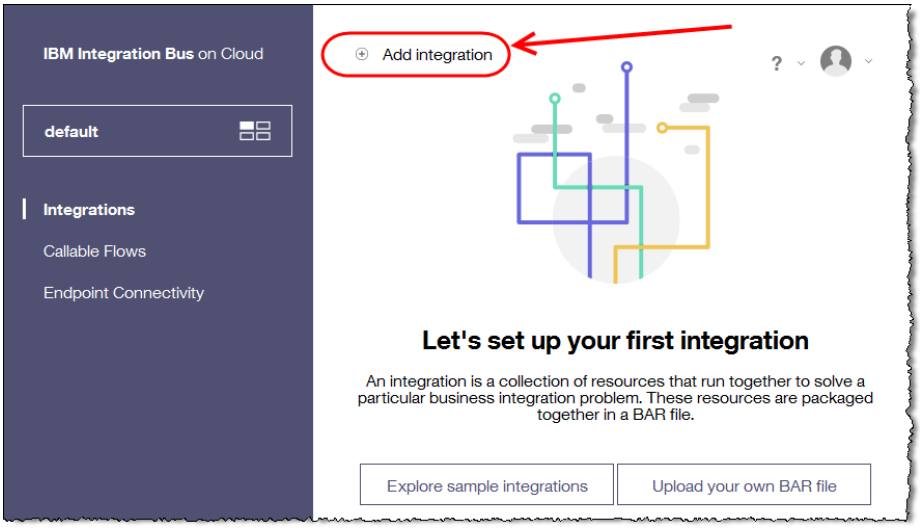
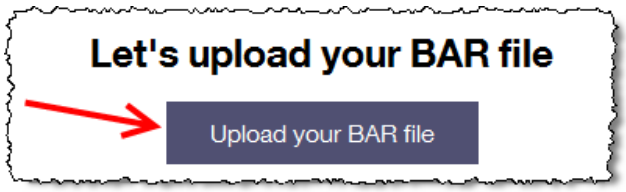
2.7 Scenario:2 Running CallableFlowInvoke in IIB on Cloud

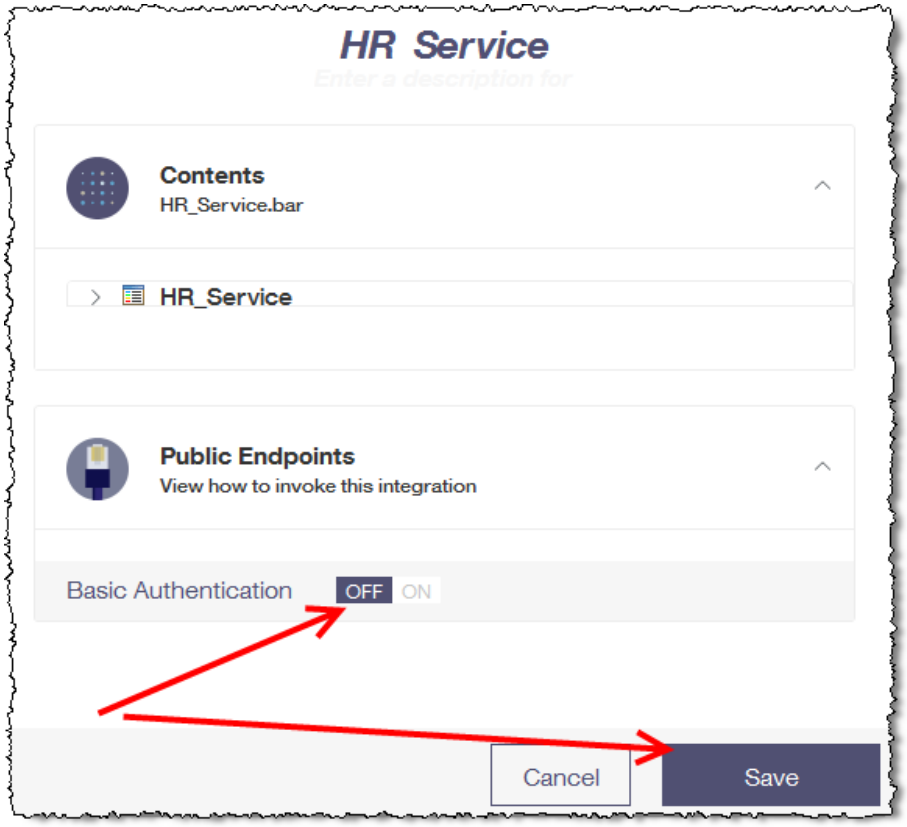
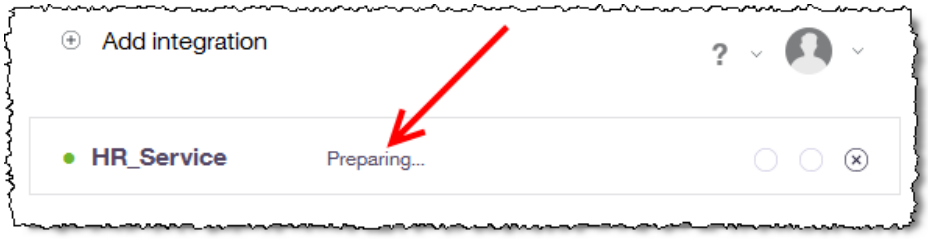


In this section of the lab you will deploy HR_Service to the IBM Managed Service IIB on Cloud environment. You will then configure the REST API running on IIB on Cloud to enable communication to your “on premises” IIB integration node TESTNODE_iibuser.

2.7.1 Deploy HR_Service to IIB on Cloud

1.	Open a new browser tab and in the Bookmarks Toolbar click “Cloud” and then the "IIB on Cloud" bookmark. For reference the URL is: https://ibm-cloud-ui.ibmintegrationbus.ibmcloud.com/
2.	Enter your ‘IBMid’ and password and click ‘**Sign in**’. 

3.	The IIB on Cloud default integration space is opened. You will now upload a BAR file containing the REST API HR_Service. Click '**Add Integration**'. 
4.	Click '**Upload your BAR file**'. 
5.	In the File Upload window, navigate to "`C:\student10\CallableflowsIC17\barfiles`" and select `HR_Service.bar`. Click Open.

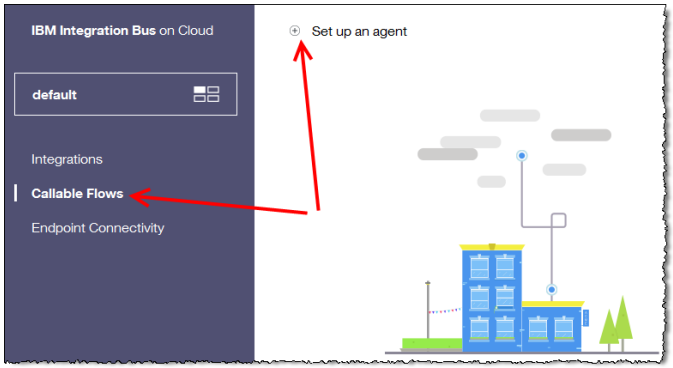
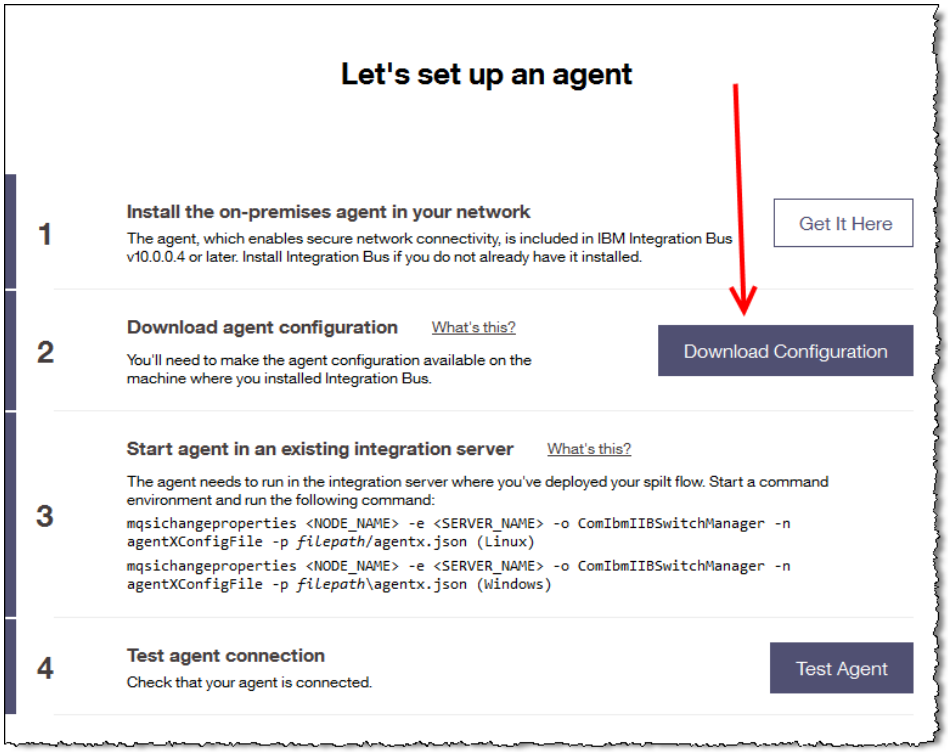
6.	IIB on Cloud will verify the contents of the bar file and present the contents as an Integration. Scroll down to the Basic Authentication section and turn off basic authentication by clicking the OFF button. Click Save. 
7.	The Web user interface will show that the integration is '**Preparing**'. This may take a minute or two to complete. 

8.	Refresh the browser session to see the status. After a few minutes the status will turn to stopped: 

2.7.2 Connect IIB on Cloud to TESTNODE_iibuser

In this part of the lab, you will create a 'switch' which will connect your IIB on Cloud system to your local IIB environment. The switch will run on IIB on Cloud. You will configure TESTNODE_iibuser to use the IIB on Cloud Switch configuration. This will allow HR_Service running on IIB on Cloud to connect via a secure connection to HR_Service_CallableFlows that is running locally on your Windows environment.

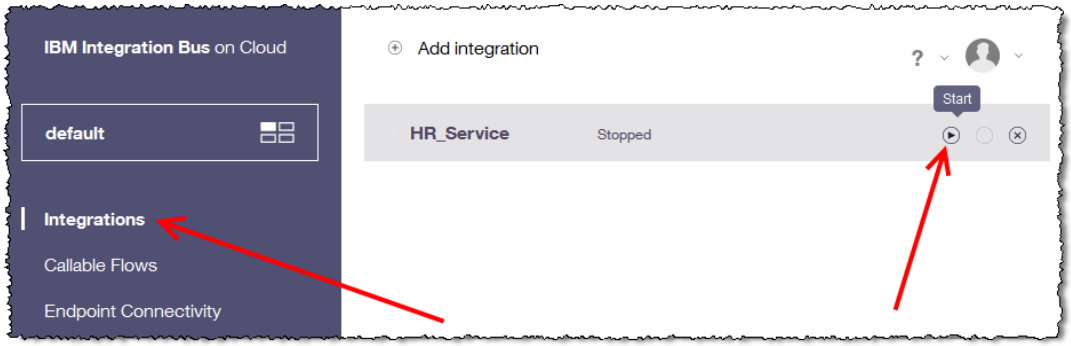
2.7.2.1 Set up an agent

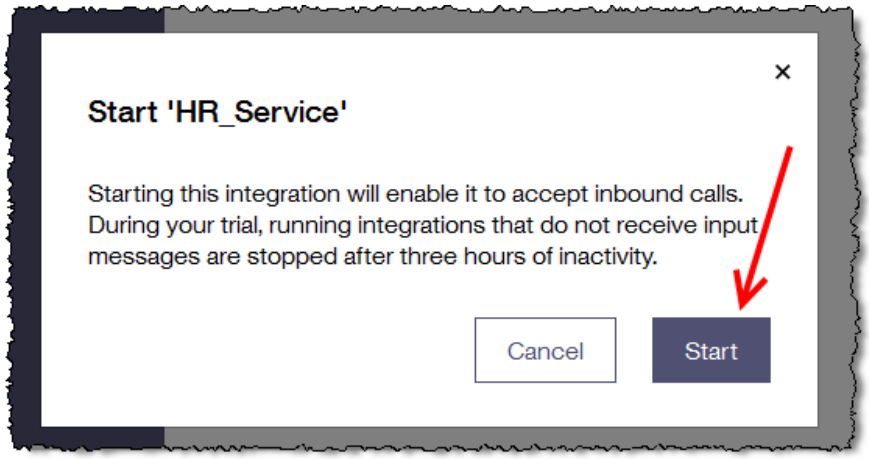
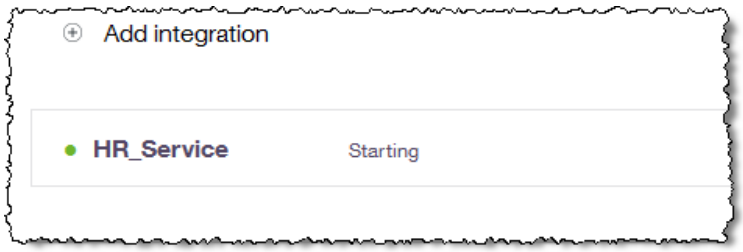
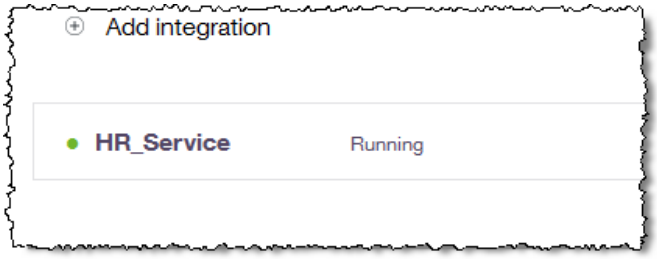
1.	Click on the **Callable Flows** tab. If you receive a message saying that the callable flow connectivity is being restarted, refresh the page. Click Set up an agent. 
2.	A window will pop-up to show the steps required for setting up the agent. You already have a local installation of IIB, so the first step is not required. Click '**Download Configuration**'. 

3.	Select to **Save** the file and then click '**OK**'. Save the file in the default location (`C:\Users\iibuser\Downloads`)
4.	To configure **TESTNODE_iibuser** to use this configuration a command file has be supplied. In an Integration Console, navigate to: "`C:\student10\CallableflowsIC17\commands`" Enter the command "`ConfigureTESTNODE_iibuserIIBonCloudAgentX.cmd`". Make sure the command responds with: `BIP8071I: Successful command completion` For information, the batch file executes this command: **`mqsichangeproperties TESTNODE_iibuser -e default -o ComIbmIIBSwitchManager -n agentXConfigFile -p C:\Users\iibuser\Downloads\agentx.json`**
5.	Now that you have configured TESTNODE_iibuser to use the IIB on Cloud agent, test the agentx configuration, by clicking on Test Agent in the IIB on Cloud browser window. 

6.	You will see the message '**1 agent connected**' in a green box:  Click **Finish**.
7.	The browser window will now show the Callable Flows that IIB on Cloud has access to on your local Windows environment: 

2.7.3 Start your IIB on Cloud integration

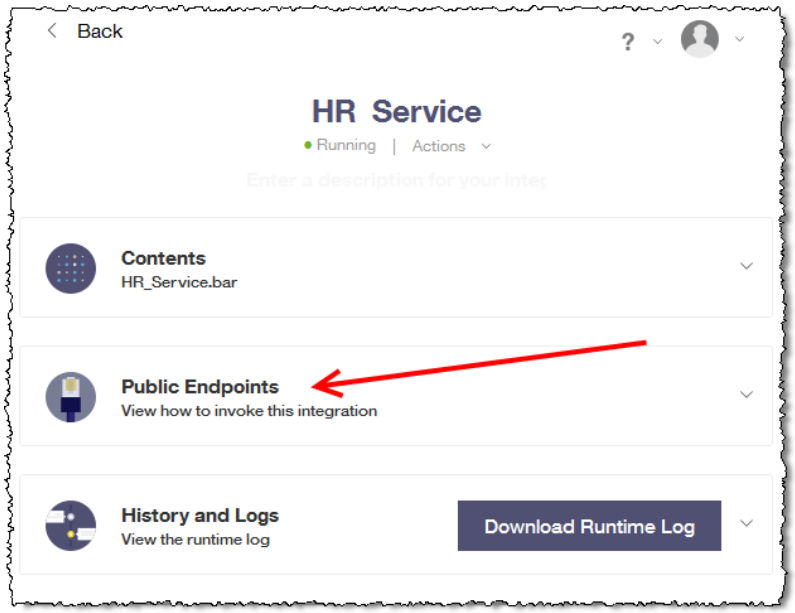
1.	In the IIB on Cloud Web UI, click **Integrations**, then click the start button for the **HR_Service** integration: 

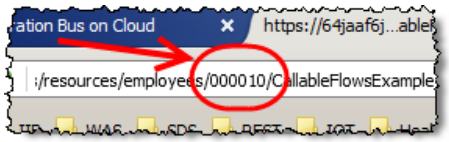
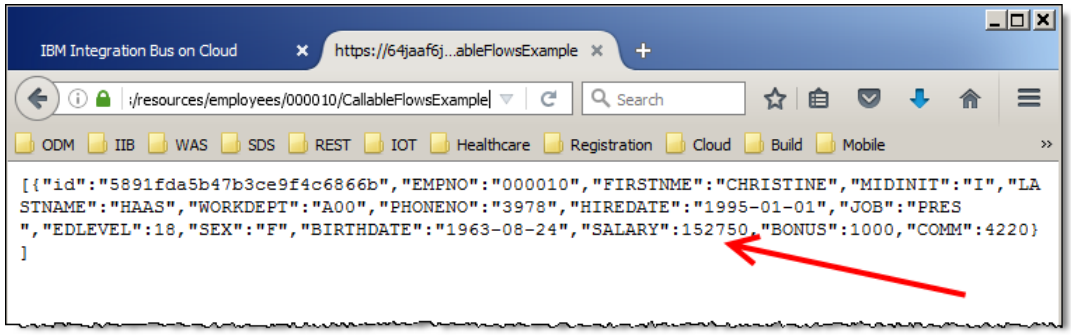
2.	Confirm that the integration will be started: 
3.	The IIB on Cloud Web UI will show that the integration is starting. It may take a minute or two for the integration to start completely and you can check its status by click on '**Refresh Listing**': 
4.	When the Integration has started, the status will change to Running: 

2.7.4 Test HR_Service running in IIB on Cloud

In this part of the lab you will test your IIB on Cloud Integration. The REST API HR_Service will call the message flow getEmployeeCallable in HR_Service_CallableFlows through the IIB Secure connection.

getEmployeeCallable will retrieve Employee data from the EMPLOYEE collection in the MongoDB database and return the response to IIB on Cloud.

1.	When HR_Service is running, click on **HR_Service** name. The view shows you more details about the integration. Click '**Public Endpoints**': 
2.	Click **HR_Service** in the "Service URLs" section, then click "Show full URL" 

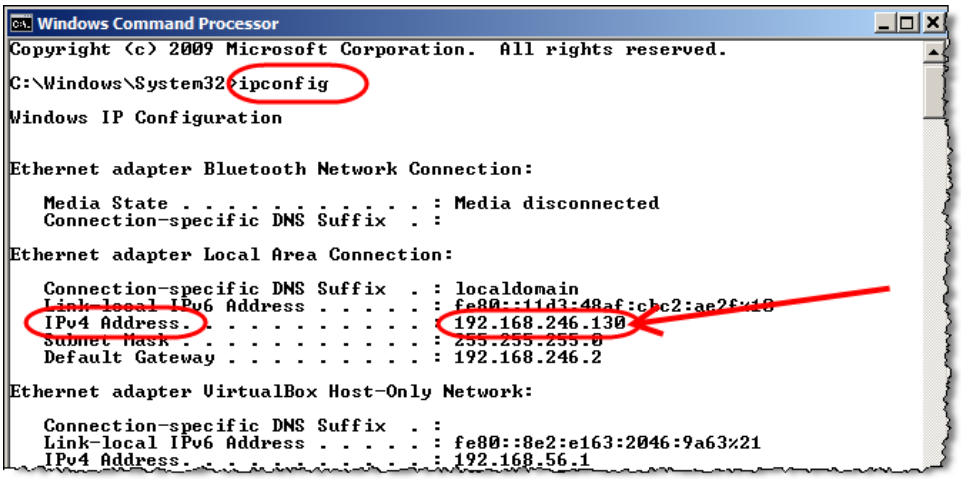
3.	The full URL will automatically be highlighted, copy the URL (ctrl c) and paste the URL into a new browser tab: 
4.	In the browser window replace `{employeeNumber}` with `000010` and press enter: 
5.	After a few seconds, the browser will show a response, indicating that the HR_Service running on IIB on Cloud can communicate successfully with the Callable Flow getEmployeeCallable running in your local Windows environment: 

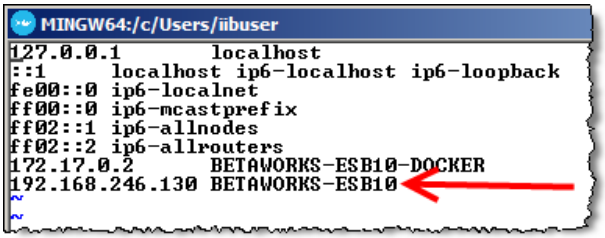
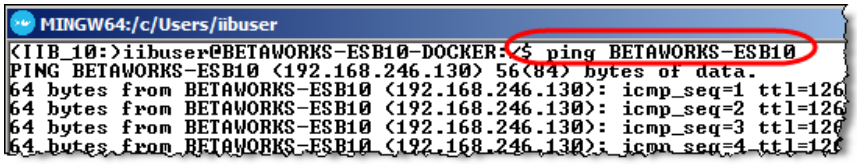
3. Appendix

3.1 Instructions if your Docker Container cannot communicate with Windows

For scenario 1 of the Callable Flows part of this lab to work correctly, IIB_Container needs to be able to successfully communicate with the Windows host BETAWORKS-ESB10.

In the lab environment you are using there is a possibility that without intervention this will not work. The following section outlines the tasks necessary to enable successful network communication between the IIB_Container running in Linux and the Windows system (where TESTNODE_iibuser is running).

1.	You will need the IP address of the Windows environment where TESTNODE_iibuser is running. The host name is BETAWORKS-ESB10. Obtain the IP address for the Windows environment by running `ipconfig` in a Windows command prompt, for example: 
2.	In the Docker Quickstart terminal (you will be in the bash shell prompt), enter: `cd /etc` then `sudo vi hosts`

3.	You will now be editing the Linux system hosts file as the administrator using vi. Follow the next steps **very carefully**: • Type '`<shift> g`' (*upper case G*) on your keyboard, this will place the cursor at the bottom of the hosts file. • Type '`o`' to add a blank line to the file - this will also put you in to "insert" mode, anything you now type appear in the file. • Type '`192.168.xxx.xxx BETAWORKS-ESB10`' where `xxx.xxx` is the last part of the exact IP address of the Windows VM. For example in the screen capture the ip address is *`192.168.246.130`*)  • Press the '**Esc**'ape key on your keyboard • Type '`:wq!`' and then press the Enter key – **note if you need to start again because you have misspelled anything, type '`:q!`' to discard your changes and start again.**
4.	If changes have been made correctly, you should now be able to ping the Windows host name (BETAWORKS-ESB10) from the IIB_Container bash command session.  Return to step 4. In the section Configure Docker Quickstart Terminal for MQSI commands on page 81.

End of Lab Guide

Note: More lab guides in this series can be found at:
https://ibm.biz/betaworks-iib